

## Success with C++

Kris Jamsa

Copyright © 1994 Jamsa Press

## Succes cu C++

Kris Jamsa

Traducere: Ion Fătu

Copyright © 1997 - Editura ALL EDUCATIONAL S.A.

ISBN 973-9229-36-0

Toate drepturile rezervate Editurii ALL EDUCATIONAL S.A.

Nici o parte din acest volum nu poate fi copiată

fără permisiunea scrisă a Editurii ALL EDUCATIONAL S.A.

Drepturile de distribuție în străinătate

aparțin în exclusivitate editurii.

Copyright © 1997 by ALL EDUCATIONAL S.A.

All rights reserved.

The distribution of this book outside Romania, without  
the written permission of ALL EDUCATIONAL S.A. is strictly prohibited.

# SUCCES CU C++

KRIS JAMSA

Traducere: Ion Fătu

Editura ALL EDUCATIONAL S.A. Bd. Timișoara nr. 58, sector 6, cod 76548  
☎ 312.11.46, 312.43.21, 311.07.44  
Fax: 311.05.65

Departamentul difuzare: ☎ 312.18.21, 311.15.47

Redactori: *prof. Viorica Fătu și Mihai Mănăstireanu*  
Tehnoredactare computerizată: *Roxana Cornelia Nistor*



058568  
B.C.U. - IASI

PRINTED IN ROMANIA

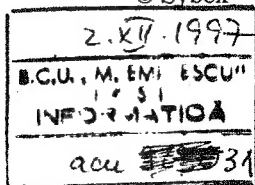


În colecția **software**

a editurii ALL EDUCATIONAL

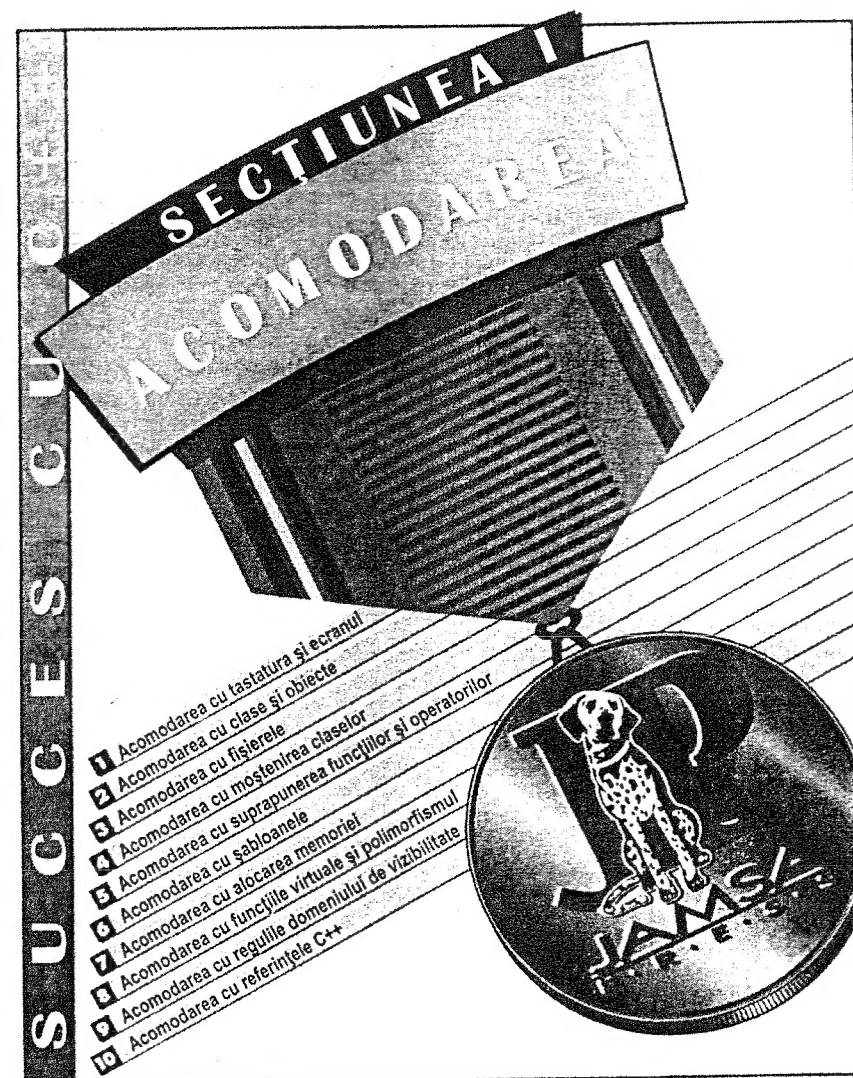
Au apărut:

1. Microsoft Access pentru Windows 95, Ghid de referință - *James E. Powel*  
© Sybex
2. Ghidul secret al calculatorului - *Russ Walter*  
© Sybex
3. Word pentru Windows 95 - ușor și rapid - *Christian Crumlish*  
© Sybex
4. Excel pentru Windows 95 - ușor și rapid - *Gerald E. Jones*  
© Sybex
5. Microsoft Excel pentru Windows 95. Ghid de referință - *Douglas E. Hergert*  
© Sybex
6. Primii pași în Internet - *Christian Crumlish*  
© Sybex



Vor apărea:

1. Biblioteca programatorului ActiveX - *Suleiman Lalani, Ramesh Chandak*  
© Jamsa Press
2. Ghidul programatorului în Web - *Kris Jamsa, Suleiman Lalani, Steve Weakley*  
© Jamsa Press
3. Biblioteca programatorului Java - *Suleiman Lalani, Kris Jamsa*  
© Jamsa Press
4. Programarea în Internet - *Kris Jamsa, Ken Cope*  
© Jamsa Press



## CUPRINS

### Secțiunea 1 - Acomodarea

#### Capitolul 1

<b>Acomodarea cu tastatura și ecranul .....</b>	<b>1</b>
Să înțelegem stream-urile I/O .....	1
Să înțelegem cin și cout .....	3
Redirectarea stream-urilor I/O .....	4
Folosirea altor stream-uri I/O .....	5
Redirijarea ieșirii programelor .....	5
Să înțelegem stream-urile I/O .....	6
Să înțelegem stream-urile de intrare și ieșire .....	6
Caractere speciale .....	6
Lucrul cu manipulatori .....	8
Stabilirea bazei de conversie .....	9
Controlul lățimii zonei de aliniere .....	9
Stabilirea caracterului de umplere .....	11
Controlul afișării valorilor în virgulă mobilă .....	12
Golirea buffer-ului .....	13
Ignorarea spațiilor albe ce preced datele de intrare .....	13
Controlul indicatorilor stream-ului I/O .....	14
Afișarea valorilor hexazecimale cu majuscule .....	15
Alinierea rezultatelor la stânga sau la dreapta .....	16
Controlul afișării cu punct fix și cu exponent .....	16
Forțarea afișării punctului zecimal .....	17
Forțarea afișării semnului unei valori .....	18
Refacerea indicatorilor stream-urilor I/O .....	18
Folosirea indicatorilor I/O .....	19
Facilități oferite de funcțiile membru ale stream-ului de intrare .....	19
Determinarea numărului de caractere extrase .....	19
Citirea unei linii de la tastatură sau de la stdin .....	20
Folosirea funcțiilor membru ale stream-ului cin .....	21
Realizarea operațiilor de intrare caracter cu caracter .....	22
Anticiparea următorului caracter la intrare .....	23
Readucerea unui caracter în buffer-ul de intrare .....	24
Detectarea sfârșitului de fișier .....	25
Ignorarea caracterelor din stream-ul de intrare .....	25
Citirea și salvarea diferitelor valori ale parametrilor I/O .....	26
Să înțelegem parametrul stream-ului de ieșire .....	27
Facilități oferite de funcțiile membru ale stream-ului de ieșire .....	27
Folosirea funcțiilor membru ale stream-ului cout .....	28
Testarea reușitei operațiilor I/O .....	28
Să înțelegem starea unui stream I/O .....	30

Intrări-ieșiri cu și fără buffer .....	30
Golirea unui buffer de ieșire .....	31
Folosirea funcțiilor membru pentru controlul indicatorilor de stream I/O .....	31
Rezumat .....	33

## Capitolul 2

<b>Acomodarea cu clase și obiecte .....</b>	<b>35</b>
Primul contact cu programarea orientată pe obiecte .....	35
Să înțelegem promovarea orientată pe obiecte .....	38
Să simplificăm definițiile unei clase .....	40
Să înțelegem moștenirea .....	40
Ingineria programelor și folosirea obiectelor .....	40
Să înțelegem obiectele și clasele .....	41
Diferența între clase și obiecte .....	42
Analiza unui exemplu complet .....	43
Când se folosesc clasele și când structurile .....	45
Accesul la membrii unei clase .....	45
Folosirea funcțiilor inline .....	47
Definirea funcțiilor membru în afara clasei .....	50
Rezolvarea conflictelor de nume între membri și parametri .....	50
Rezolvarea conflictelor de nume .....	51
Să înțelegem membrii de tip private ai unei clase .....	51
Să înțelegem membrii de tip private ai unei clase .....	54
Să înțelegem camuflajul informației .....	55
Să înțelegem constructorii și destructorii unei clase .....	55
Să înțelegem funcțiile constructor .....	57
Să înțelegem funcțiile destructor .....	58
Să înțelegem funcțiile destructor .....	60
Folosirea funcțiilor constructor multiple .....	60
Folosirea argumentelor prestabilite la funcțiile constructor .....	61
Un alt mod de a inițializa membrii unei clase .....	62
Atribuirea valorii unui obiect altui obiect .....	63
Obiecte și funcții .....	65
Folosirea obiectelor și a funcțiilor .....	66
Să înțelegem membrii unei clase .....	67
Folosirea unui tablou de clase .....	68
Rezumat .....	70

## Capitolul 3

<b>Acomodarea cu fișierele .....</b>	<b>73</b>
Să înțelegem operațiile I/O cu fișiere .....	73
Deschiderea unui fișier pentru ieșire .....	74
Deschiderea unui fișier pentru ieșire .....	75
Folosirea funcției constructor pentru deschiderea unui fișier .....	75
Folosirea manipulatorilor și a funcțiilor membru de ieșire .....	76

Realizarea operațiilor de scriere formatată a fișierelor .....	77
Scrierea formatată a fișierelor .....	78
Controlul modului de deschidere a fișierelor de ieșire .....	78
Caractere speciale și fișiere I/O .....	79
Alți specificatori ai modului de deschidere a fișierelor .....	79
Controlul operațiilor pe fișiere .....	80
Deschiderea fișierelor pentru operații de intrare .....	81
Realizarea operațiilor de intrare pe fișiere .....	81
Deschiderea unui fișier pentru intrare .....	83
Testarea reușitei unei operații I/O .....	84
Realizarea operațiilor pe fișiere binare .....	86
Realizarea operațiilor pe fișiere binare .....	89
Atenție la operațiile de inserție/extracție cu fișiere binare .....	89
Deschiderea fișierelor pentru operații de citire și scriere .....	92
Fișiere cu acces aleator .....	92
Recapitularea modurilor de deschidere a unui fișier .....	94
Să înțelegem fișierele cu acces aleator .....	97
Realizarea ieșirii la imprimantă .....	97
Rezumat .....	98

## Capitolul 4

<b>Acomodarea cu moștenirea claselor .....</b>	<b>101</b>
Conceptul de moștenire .....	101
Analiza unui exemplu simplu .....	102
Să înțelegem moștenirea claselor .....	106
Folosirea funcțiilor constructor la moștenirea claselor .....	107
Analiza unui exemplu .....	110
Să înțelegem cuvântul-cheie public .....	112
Analiza altui exemplu .....	112
Un ultim exemplu de moștenire pe un singur nivel .....	116
Prototipuri, revizii, finalizări .....	121
Să înțelegem moștenirea multiplă .....	122
Să înțelegem moștenirea multiplă .....	125
Să înțelegem moștenirea pe mai multe nivele .....	125
Să înțelegem membrii de tip protected ai unei clase .....	128
Să înțelegem membrii protejați ai unei clase .....	131
Rezumat .....	131

## Capitolul 5

<b>Acomodarea cu suprapunerea funcțiilor și operatorilor .....</b>	<b>133</b>
Să înțelegem suprapunerea funcțiilor .....	134
Să înțelegem suprapunerea funcțiilor .....	135
Analiza altor exemple .....	135
Folosirea parametrilor prestabiliți .....	141
Să înțelegem parametrii prestabiliți .....	143



Cum să creăm propriile biblioteci de funcții .....	144
Să înțelegem suprapunerea operatorilor .....	144
Să înțelegem suprapunerea operatorilor .....	149
Reguli pentru suprapunerea operatorilor .....	149
Suprapunerea operatorilor I/O .....	150
Analiza altor exemple .....	153
Rezumat .....	157

## Capitolul 6

<b>Acomodarea cu șabloanele .....</b>	<b>159</b>
Crearea primului șablon de funcție .....	160
Să înțelegem șabloanele de funcții .....	163
Clarificarea definiției unui șablon .....	165
Șabloane care folosesc mai multe tipuri .....	165
Șabloane care admit mai multe tipuri .....	168
Unde se plasează șabloanele .....	168
Crearea șabloanelor de clase .....	168
Analiza unui alt exemplu .....	172
Rezumat .....	177

## Capitolul 7

<b>Acomodarea cu alocarea memoriei .....</b>	<b>179</b>
Un exemplu simplu de alocare a memoriei .....	179
Lucrul cu memoria dinamică .....	183
Alocarea dinamică a memoriei în C++ .....	184
Alocarea dinamică a memoriei folosind operatorul new .....	186
Lucrul cu variabile pointer .....	187
Inițializarea unei valori folosind operatorul new .....	189
Să înțelegem aritmetica pointerilor .....	190
Tratarea pointerilor ca tablouri .....	191
Utilizarea pointerilor cu parametri de funcții .....	192
Folosirea pointerilor la structuri și clase .....	194
Pointerii nu sunt referințe .....	195
Atenție la pointeri și la variabile locale .....	196
Funcții care returnează pointeri .....	197
Spațiul liber de memorie .....	198
Condițiile de memorie insuficientă .....	198
Specificarea unei funcții pentru tratarea erorilor de memorie insuficientă .....	192
Suprapunerea operatorilor new și delete .....	201
Rezumat .....	206

## Capitolul 8

<b>Acomodarea cu funcțiile virtuale și polimorfismul .....</b>	<b>209</b>
Să înțelegem funcțiile virtuale .....	209

Să înțelegem funcțiile virtuale .....	212
Să înțelegem polimorfismul .....	214
Ce nu este polimorfismul .....	214
Folosirea funcțiilor membru ale clasei de bază .....	216
Reguli pentru funcții polimorfice .....	220
Funcții virtuale și moștenirea pe mai multe nivele .....	221
Să înțelegem funcțiile pur virtuale .....	224
Să înțelegem funcțiile pur virtuale .....	226
Rezumat .....	226

## Capitolul 9

<b>Acomodarea cu regulile domeniului de vizibilitate .....</b>	<b>227</b>
Declarații și definiții .....	227
Principii de bază ale domeniului de vizibilitate .....	228
Să înțelegem declarațiile și definițiile .....	229
Să înțelegem domeniul local .....	229
Declaraarea variabilelor în interiorul unui bloc de instrucțiuni .....	231
Să înțelegem domeniul local .....	232
Să înțelegem domeniul unei funcții .....	232
Să înțelegem domeniul unei funcții .....	233
Să înțelegem domeniul unui fișier .....	233
Să înțelegem domeniul unui fișier .....	234
Să înțelegem domeniul unei clase .....	234
Să înțelegem membrii de tip public și private ai unei clase .....	234
Folosirea membrilor de tip public și private ai unei clase .....	234
Să înțelegem accesul controlat .....	236
Atenție la membrii ascunși .....	239
Să înțelegem clasele friend .....	239
Accesul special al claselor friend la membrii unui obiect .....	242
Să înțelegem membrii de tip protected .....	243
Folosirea membrilor de tip protected .....	245
Efectul calificatorilor asupra domeniului .....	245
Folosirea calificatorului extern .....	245
Inițializarea unei variabile externe .....	246
Să înțelegem membrii de tip static ai unei clase .....	247
Partajarea unei variabile membru a unui obiect .....	249
Funcții membru în interiorul și în exteriorul definiției clasei .....	249
Partajarea codului unei funcții membru .....	250
Să revedem operatorul de rezoluție globală .....	250
Să înțelegem durata de viață a unui identificator .....	252
Să înțelegem editarea legăturilor .....	254
Rezumat .....	259

<b>Capitolul 10</b>	
<b>Acomodarea cu referințele C++</b>	<b>261</b>
O referință este un nume asociat	261
Crearea unei referințe	263
O referință nu este o variabilă	264
Posibile sintaxe pentru referințe	265
Reguli pentru lucrul cu referințe	265
Referința este o valoare sau o adresă?	267
Folosirea referințelor drept parametri	267
Folosirea referințelor cu parametri și obiectele unei structuri	270
Explicați operațiile cu referințe	272
Folosirea referințelor la obiecte	274
Atenție la obiectele ascunse	276
Rezumat	277

## Secțiunea 2: Aprofundarea

<b>Capitolul 11</b>	
<b>Aprofundarea stream-urilor I/O din C++</b>	<b>281</b>
Să înțelegem relația între clasele stream-urilor I/O	281
Să înțelegem stream-urile I/O	284
Crearea propriilor manipulatori I/O	284
Crearea propriilor manipulatori	285
Manipulatori cu parametri multipli	289
Manipulatori cu parametri	291
Crearea unui fișier de manipulatori proprii	293
Să înțelegem stream-urile I/O asociate	293
Mai multe despre interacțiunea claselor de stream-uri	295
Să înțelegem ieșirea formatată și neformatată	295
Să înțelegem intrările/ieșirile prin buffer	297
Deschiderea unui fișier pentru operații I/O prin buffer	299
Imagine de ansamblu	301
Rezumat	301

<b>Capitolul 12</b>	
<b>Aprofundarea stream-urilor șir</b>	<b>303</b>
Să analizăm fișierul STRSTREAM.H	303
Folosirea stream-urilor șir de intrare	304
Să înțelegem stream-urile șir de intrare	306
Citirea mai multor valori dintr-un stream șir de intrare	307
Citirea mai multor valori dintr-un stream șir de ieșire	311
Folosirea funcțiilor membru ale stream-ului șir de intrare	311
Folosirea stream-urilor șir de ieșire	312

Folosirea unui buffer dinamic	315
Crearea unui stream șir de ieșire dinamic	316
Folosirea funcțiilor membru ale stream-ului șir de ieșire	316
Blocarea unui buffer dinamic prin funcția str	318
Stream-uri șir de intrare/ieșire	318
Să înțelegem relațiile dintre stream-uri	320
Rezumat	320

<b>Capitolul 13</b>	
<b>Aprofundarea funcțiilor virtuale</b>	<b>323</b>
Analiza unui exemplu simplu	323
Polimorfismul nu înseamnă suprapunerea funcțiilor	325
Să înțelegem legăturile dinamice	327
Să înțelegem tabelele de funcții virtuale	328
Suprasolicitarea generată de funcțiile virtuale	329
Să înțelegem destructorii virtuali	330
Rezumat	332

<b>Capitolul 14</b>	
<b>Aprofundarea tratării excepțiilor</b>	<b>335</b>
Ce sunt cazurile de excepție	336
Preluarea excepțiilor într-un program	337
Cum se denumesc excepțiile	337
Denumirea și generarea excepțiilor	339
Excepții locale ale unei clase	339
Tratarea excepțiilor	342
Informații suplimentare despre o excepție	343
Returnarea unor valori alături de excepție	348
Preluarea mai multor excepții într-o singură instrucțiune catch	348
Ce se întâmplă la lansarea unei excepții	349
Excepțiile întrerup lanțul de apel al funcțiilor	351
Atenție la excepțiile subprogramelor de bibliotecă	351
Specificarea interfeței pentru excepții	352
Excepții pe mai multe niveluri	352
Repetarea unei operații	353
Tratarea excepțiilor nepreluare	354
Rezumat	357

<b>Capitolul 15</b>	
<b>Aprofundarea gestiunii zonei de memorie alocabilă</b>	<b>359</b>
Examinarea zonei de memorie alocabilă	359
Evidența listei cu legături a zonei alocabile	364
Testarea validității unei intrări din zona alocabilă	364
Să înțelegem starea zonei alocabile	366

O verificare rapidă a zonei alocabile .....	367
Validarea unei anumite intrări din zona alocabilă .....	368
Testarea spațiului de memorie disponibil din zona alocabilă .....	369
Compactarea spațiului de memorie disponibil .....	371
Rezumat .....	373

## Secțiunea 3 - Sinteze

### Capitolul 16

<b>Să construim clase cu operatori I/O intrinseci .....</b>	<b>377</b>
Intrări/ieșiri cu membrii clasei în stil clasic .....	377
Să înțelegem clasele cu operatori I/O intrinseci .....	381
Crearea unei clase cu operatori I/O intrinseci .....	381
Reguli pentru crearea de clase cu operatori I/O intrinseci .....	384
Operații de intrare pe stream .....	384
Operații I/O pe stream fișier .....	385
Atenție la fișierele binare .....	387
Rezumat .....	390

### Capitolul 17

<b>Crearea unei biblioteci de clase .....</b>	<b>391</b>
Crearea unei biblioteci de clase .....	391
Crearea unei biblioteci de obiecte .....	393
Crearea unei biblioteci de clase .....	394
Rezumat .....	399

### Capitolul 18

<b>Crearea unei clase de liste dinamice .....</b>	<b>401</b>
Analiza unei liste cu legături simple .....	401
Să înțelegem funcția <code>append_node</code> .....	403
Folosirea unui șablon de clasă .....	406
Când programele acceptă liste cu legături duble .....	407
Adăugarea unor membri ai clasei .....	412
Șabloane și biblioteci de clase .....	415
Rezumat .....	416

<b>Index .....</b>	<b>417</b>
--------------------	------------

## CAPITOLUL 1

# ACOMODAREA CU TASTATURA ȘI ECRANUL

Fiecare program pe care îl creai, indiferent de scopul său, va conține operații de intrare/ieșire. Acest capitol analizează în detaliu intrarea de la tastatură și ieșirea pe ecranul monitorului. După terminarea acestui capitol, vei cunoaște:

- Ce este un stream I/O
- Cum folosește compilatorul un fișier antet
- De ce sunt importante *cin* și *cout*
- Cum se poate formata intrarea și ieșirea
- Ce este un manipulator
- Câteva utilizări ale funcțiilor componente ale stream-urilor de intrare și ieșire
- Cum se realizează o ieșire prin buffer (memorie tampon)

Dacă sunteți familiarizat cu operațiile I/O din C++, se recomandă să revedeți, totuși, manipulatorii prezentați în acest capitol. Capitolul 11 examinează clasele I/O în detaliu, iar până atunci vei face deja cunoștință cu clasele și metodele I/O. Dacă sunteți începători în C++, experimentați fiecare din programele prezentate aici; modificați-le și observați rezultatele obținute.

## SĂ ÎNȚELEGEM STREAM-URILE I/O

În sensul cel mai simplu, un stream I/O este o secvență de caractere afișate pe ecranul monitorului sau citite de la tastatură. Pentru operațiile standard de intrare/ieșire, în C++ se folosesc stream-urile *cin* (pentru intrare) și *cout* (pentru ieșire). După cum se va vedea, C++ definește stream-urile I/O prin clase. Definiția acestora apare în fișierele antet `IOSTREAM.H` sau `ISTREAM.H` sau `OSTREAM.H`, în funcție de compilatorul pe care îl folosești. Dacă examinați conținutul acestor fișiere, veți vedea declarațiile pentru diferite metode și operatori de intrare/ieșire. Este chiar indicat să tipăriți conținutul acestor fișiere pentru a-l folosi ca referință. Nu fiți îngrijorați dacă nu înțelegeți toate declarațiile și definițiile ce apar în aceste fișiere; le vom examina în detaliu în Capitolul 11.

**Observație:** Drept exemplu de diferențe între compilatoare, Borland C++ folosește *IOSTREAM.H* pentru majoritatea definițiilor legate de stream-uri, pe când compilatorul Microsoft Visual C++ folosește pentru aceste definiții fișierele *IOS.H*, *ISTREAM.H*, *OSTREAM.H* și *STREAMB.H*.

De obicei, programul C++ va folosi fișierul *IOSTREAM.H* drept una din primele instrucțiuni. Următorul program, *SUCCESS.CPP*, folosește **cout** pentru a afișa un mesaj pe ecran:

```
#include <iostream.h>

void main(void)
{
    cout << "Getting up to speed with C++ I/O";
}
```

După cum se vede, programul folosește operatorul de ieșire sau de inserție (semnul <<) pentru a afișa mesaje pe ecran. În capitolul 11 veți învăța modul de definire a operatorului de inserție C++ pentru clasele de stream-uri I/O. Tot în acel capitol veți descoperi cum se definește **cout** ca un obiect.

Următorul program, *GET\_NAME.CPP*, cere utilizatorului să-și introducă prenumele, apoi folosește operatorul de intrare sau de extragere (semnul >>) pentru a atribui variabilei *first\_name* literele introduse:

```
#include <iostream.h>

void main(void)
{
    char first_name[64];

    cout << "Type in your first name: ";

    cin >> first_name;

    cout << "Hello, " << first_name;
}
```

Programul folosește operatorul de ieșire pentru a afișa solicitarea de introducere a numelui utilizatorului, după care, folosind operatorul de intrare, atribuie variabilei *first\_name* literele tastate. Apoi, folosește încă o dată operatorul de ieșire pentru a afișa mesajul „Hello”, urmat de numele introdus:

```
cout << "Hello, " << first_name;
```

Când tipăriți informații folosind operatorii de extracție multiplă în acest mod, C++ afișează informația așa cum apare în instrucțiune, de la stânga spre dreapta. Rulați programul anterior și introduceți prenumele, urmat apoi de numele dvs. Veți vedea că programul afișează numai prenumele, indiferent de informa-

ția introdusă. Când folosiți *cin* pentru a citi șiruri de caractere (cum ar fi un nume) de la tastatură, *cin* folosește spațiile albe (spațiu, sau tab, sau sfârșit de linie) pentru a separa șirurile. Dacă aveți nevoie să citiți o linie de text de la tastatură, puteți folosi funcția membru *cin.getline*, ce va fi prezentată puțin mai târziu. Următorul program, *FULLNAME.CPP*, plasează două variabile în instrucțiunea de intrare pentru a citi atât prenumele, cât și numele utilizatorului:

```
#include <iostream.h>

void main(void)
{
    char first_name[64];
    char last_name[64];

    cout << "Type your first and last names: ";
    cin >> first_name >> last_name;

    cout << "Hello, " << first_name << " " << last_name;
}
```

Programul folosește următoarea instrucțiune pentru a citi valorile la două variabile:

```
cin >> first_name >> last_name;
```

Când folosiți doi sau mai mulți operatori de extragere în aceeași instrucțiune, C++ atribuie variabilelor valorile de la intrare, începând de la stânga spre dreapta. Observați, de asemenea, folosirea ghilimelelor în mesajul de ieșire, pentru a separa prenumele de nume. Dacă îndepărtați ghilimelele, numele va fi scris, fără pauză, în continuarea prenumelui.

```
cout << "Hello, " << first_name << " " << last_name;
```



### SĂ ÎNȚELEGEM CIN ȘI COUT

Un stream I/O este o secvență de caractere, pentru ieșire sau pentru intrare. Când un program trebuie să afișeze rezultate pe ecranul monitorului (sau pe dispozitivul standard de ieșire), acesta trebuie să insereze caracterele în stream-ul de ieșire **cout** folosind operatorul de inserție astfel:

```
cout << "Success with C++" << endl;
```

Când un program primește intrare de la tastatură (sau de la un dispozitiv standard de intrare), acesta extrage caractere din stream-ul de intrare **cin**, folosind operatorul de extragere sub forma:

```
cin << nume_variabila;
```

## REDIRECTAREA STREAM-URILOR I/O

C++ asociază stream-urile *cin* și *cout* cu dispozitive standard de intrare și ieșire ale sistemului de operare. În acest mod, se pot folosi stream-urile I/O pentru a scrie programe ce redirează intrarea și ieșirea. De exemplu, următoarea linie de comandă cere sistemului DOS (sau UNIX) să trimită ieșirea programului SUCCESS spre un fișier numit OUTPUT.DAT, și nu pe ecranul monitorului:

```
C:\> SUCCESS > OUTPUT.DAT <ENTER>
```

Următorul program, LINE\_NBR.CPP, afișează un număr în fața fiecărei linii a intrării redirijate.

```
#include <iostream.h>

void main(void)
{
    char line[256];
    long line_number = 0;

    while (!cin.eof())                // Loop until no characters
                                    // are available
    {
        cout << ++line_number << '\t'; // Display a line number
                                    // and a tab
        cin.getline(line, sizeof(line)); // Read a line of text
        cout << line << endl;          // Display the line of text
    }
}
```

Programul folosește stream-urile de intrare/ieșire *cin* și *cout*. Totuși, așa cum se observă, acesta beneficiază de existența a două funcții membru, *cin.eof* și *cin.getline*. Să ne reamintim că *cin* și *cout* sunt obiecte, deci definesc funcții membre. Dacă n-ați folosit niciodată aceste două funcții nu vă faceți griji. Le vom prezenta mai târziu, în această lecție. Pe lângă folosirea acestor două funcții, programul folosește manipulatorul *endl* pentru a scrie caracterul newline (linie nouă), compus din caracterele carriage-return (retur de car) și line-feed (avans hârtie).

Pentru a afișa numărul fiecărei linii a fișierului SUCCESS.CPP, de exemplu, se poate utiliza LINE\_NBR astfel:

```
C:\> LINE_NBR < SUCCESS.CPP <ENTER>
1      #include <iostream.h>
2
3      void main(void)
4      {
5          cout << "Getting up to speed with C++ I/O";
6      }
```

Dacă se dorește salvarea rezultatelor pe un fișier (de exemplu, pe fișierul OUTPUT.DAT), se poate redireja ieșirea programului LINE\_NBR astfel:


```
C:\> LINE_NBR <SUCCESS.CPP> OUTPUT.DAT <Enter>
```

## FOLOSIREA ALTOR STREAM-URI I/O

Majoritatea programelor C++ pe care le veți întâlni folosesc stream-urile de I/O *cin* și *cout*. Așa cum se va vedea, fișierul IOSTREAM.H definește și stream-urile I/O *cerr* și *clog*. Tabela 1.1. descrie pe scurt diferite stream-uri I/O:

Stream	Scop	Exemplu
cin	intrare tastatură (stdin)	cin >> name;
cout	ieșire ecran (stdout)	cout << "Hello, world!";
cerr	ieșire dispozitiv erori standard (stderr)	cerr << "Eroare critică..;
clog	ieșire buffer-izată pe dispozitiv erori standard (stderr)	clog << "Mesaj eroare..;

Tabela 1.1. Rezumat al stream-urilor I/O din C++



### REDIRIJAREA IEȘIRII PROGRAMELOR

Când se rulează un program, C++ asociază stream-urile I/O *cin* și *cout* cu dispozitivele standard de intrare, respectiv ieșire. În mod prestabilit, sistemul de operare asimilează dispozitivul standard de intrare cu tastatura, iar dispozitivul standard de ieșire cu ecranul monitorului. Dacă se apelează un program C++ folosind operatorii de redirejare I/O în linia de comandă, se poate redireja intrarea și ieșirea programului. Cu alte cuvinte, se poate redireja stream-ul de ieșire *cout* de la ecran spre un fișier sau un alt program. În mod asemănător, se poate redireja stream-ul de intrare *cin* de la tastatură spre un fișier.

Când un program caută și procesează erorile critice, mesajele de eroare nu pot fi redirijate de la ecranul monitorului. În acest caz trebuie folosit stream-ul *cerr* pentru afișarea mesajelor. De exemplu, următorul program, USE\_CERR\_CPP afișează un mesaj folosind *cerr*:

```
#include <iostream.h>

void main(void)
{
    cerr << "You cannot redirect this message";
}
```

Încercați să compilați și să executați acest program, folosind operatorii de redirejare ai sistemului de operare. Veți vedea că nu este posibilă redirejarea



mesajelor scrise cu *cerr*. Mai târziu, tot în acest capitol, veți examina ieșirea prin buffer folosită de *cout* și *clog*.



### SĂ ÎNȚELEM STREAM-URILE I/O

În cel mai simplu sens, un stream I/O este o succesiune de caractere, scrise pe ecran sau citite de la tastatură. Fișierul antet IOSTREAM.H definește patru stream-uri I/O: *cout*, *cin*, *cerr* și *clog*. Stream-ul *cout* permite afișarea rezultatelor pe ecran sau pe dispozitivul de ieșire standard. În mod asemănător, *cin* realizează citirea caracterelor de la tastatură sau de la dispozitivul standard de intrare. În sfârșit, *cerr* și *clog* determină programele să afișeze ieșirea pe dispozitivul de eroare standard. Pentru a transmite ieșirea spre un stream I/O, se folosește operatorul de extragere (<<), ca mai jos:

```
cout << "Hello, successful world!";
cerr << "Mesaj de eroare";
```

Pentru a obține intrare de la *cin*, se folosește operatorul de inserție (>>), ca mai jos:

```
cin >> nume;
```

C++ permite scrierea mai multor operatori sau valori de date în stream-ul de ieșire sau a unor variabile în stream-ul de intrare:

```
cout << "introduceți numele și prenumele" << endl;
cin >> nume >> prenume;
```

### SĂ ÎNȚELEM STREAM-URILE DE INTRARE ȘI IEȘIRE

Mulți dintre manipulatorii și funcțiile discutate în acest capitol corespund stream-urilor de intrare și ieșire. La examinarea fișierului IOSTREAM.H veți găsi unele definiții ce folosesc *ios* și altele ce folosesc *istream* și *ostream*. Definițiile ce folosesc *ios* corespund stream-urilor intrare și ieșire (de aceea apare prescurtarea io). În mod similar, definițiile bazate pe *istream* corespund stream-urilor de intrare, iar cele bazate pe *ostream* - stream-urile de ieșire. Prin examinarea definițiilor stream-urilor se poate înțelege mai bine fiecare manipulator sau funcție membru. În Capitolul 11 se vor trata mai detaliat relațiile între clase.

### CARACTERE SPECIALE

Unul din programele precedente a folosit manipulatorul *endl* pentru a genera o secvență CR/LF (retur de car. - avans hârtie). În mod asemănător, următorul program, THREELIN.CPP, folosește un manipulator *endl* pentru a afișa ieșirea pe trei linii.

```
#include <iostream.h>
```

```
void main(void)
```

```
{
    cout << "Success," << endl;
    cout << "with," << endl;
    cout << "C++!" << endl;
}
```

Manipulatorul *endl* se poate folosi de mai multe ori pe aceeași linie, ca în programul ce urmează, THREEOO.CPP, ce este funcțional identic cu cel precedent:

```
#include <iostream.h>
```

```
void main(void)
```

```
{
    cout << "Success" << endl << "with" << endl << "C++!" << endl;
}
```

În mod similar, un alt program prezentat anterior în acest capitol folosea caracterele *\t* pentru a genera tabulatori. Când se realizează ieșirea pe *cout*, *cerr* sau *clog* în stream-ul de ieșire se pot folosi caracterele speciale prezentate în Tabela 1.2.

Caracterul	Semnificația
\a	Alertă sau clopoțel
\b	Backspace
\f	Început de pagină
\n	Început de linie nouă (echivalent cu <i>endl</i> )
\r	Carriage return (începutul liniei curente) fără avans hârtie (linefeed)
\t	Tabulator orizontal
\v	Tabulator vertical
\\	Backslash
\?	Semn de întrebare
\'	Apostrof
\"	Ghilimele
\0	Caracter nul
\ooo	Valoare în octal, ca \033
\xhhh	Valoare hexazecimală, ca \x1B

Tabela 1.2. Caractere speciale folosite cu *cout*, *cerr* și *clog*

Următorul program, SPECCHAR.CPP, folosește aceste simboluri speciale pentru a șterge ecranul și apoi pentru a emite un semnal sonor prin difuzorul calculatorului. Pentru ștergerea ecranului, trebuie să fie încărcat driver-ul de dispozitiv ANSI. Programul folosește secvența specială ANSI **Esc[2J** pentru a șterge ecranul:

```
#include <iostream.h>

void main(void)
{
    // Clear the screen display
    cout << "\033[2J";

    // Sound the computer's speaker
    cout << "Beep\\a\\tBeep\\a\\t\\Beep\\a";
}
```

Programul folosește caracterul `\a` pentru semnalul sonor și caracterul `\t` pentru avansul cursorului cu următorul spațiu de tabulare. Multe din programele prezentate în această carte vor folosi diferite caractere speciale.

Caracterele speciale pot fi folosite cu fiecare din stream-urile de ieșire, la fel și cu stream-urile de fișier din Capitolul 3. De exemplu, următorul program, BEEPCERR.CPP, folosește caracterul `\a` pentru emiterea semnalului sonor înainte de afișarea unui mesaj către *cerr*:

```
#include <iostream.h>

void main(void)
{
    cerr << "\\aSome error message\\t\\tXXX-11-XXXX";
}
```

După cum se poate vedea, programul folosește caracterul `\a` pentru semnalul sonor și caracterul `\t` pentru a genera un tabulator. În Capitolul 11 veți învăța cum să creați proprii dumneavoastră manipulatori ce corespund acestor caractere speciale.

## \*LUCRUL CU MANIPULATORI

Un *manipulator* de stream I/O este un element prin care se filtrează ieșirea unui stream I/O. De exemplu, manipulatorii *hex*, *oct* și *dec* determină afișarea unei valori în hexazecimal, octal sau zecimal. Următorul program, MANIPULA.CPP, ilustrează modul în care sunt folosiți acești manipulatori.

```
#include <iostream.h>

void main(void)
```

```
int number = 1001;

cout << "Decimal: " << number << "\\tHexadecimal: " << hex <<
    number << endl;
cout << "Decimal: " << dec << number << "\\tOctal: " << oct <<
    number << endl;
```

Dacă examinați a treia instrucțiune a programului, veți observa folosirea manipulatorului *dec*. Ar putea părea surprinzător de ce este necesar să folosim manipulatorul cu valoarea zecimală 1001. Așa cum se va vedea, stream-ul de ieșire *cout* afișează valorile, în mod prestabilit, în zecimal. Dacă programul folosește manipulatorii *oct* sau *hex*, *cout* va afișa valorile în octal, respectiv în hexazecimal, în funcție de ultimul manipulator folosit. De aceea, pentru a tipări valoarea 1001 în zecimal, programul trebuie să folosească manipulatorul *dec*. Experimentați programul, adăugând eventual următoarea instrucțiune la sfârșitul acestuia:

```
cout << "Afișarea valorii 1001:" << 1001;
```

Deoarece manipulatorul *oct* a fost ultimul folosit în program, *cout* va afișa valoarea în octal, nu în zecimal, așa cum poate părea la prima vedere.



### STABILIREA BAZEI DE CONVERSIE

Manipulatorii *dec*, *hex* și *oct* ajută la stabilirea bazei de conversie pe care o va folosi un stream de ieșire pentru afișarea numerelor. De exemplu, următoarea instrucțiune dirijează stream-ul *cout* să tipărească valoarea 255 în hexazecimal:

```
cout << hex << 255;
```

După execuția instrucțiunii, totuși, stream-ul *cout* va folosi baza hexazecimală pentru toate ieșirile de tip numeric, până se va selecta o nouă bază. Nu este obligatoriu să tipăriți o valoare pentru a schimba baza. Următoarea instrucțiune, de exemplu, dirijează *cout* pentru a afișa ulterior valori în zecimal:

```
cout << dec;
```

În sfârșit, când se specifică o bază folosind manipulatorii *dec*, *hex* sau *oct*, această bază va fi specifică unui stream I/O. Bazele celorlalte stream-uri nu vor fi afectate.

Pe lângă manipulatorii *dex*, *hex* și *oct* de stabilire a bazei de conversie, într-un program se poate folosi și manipulatorul *setbase* care este definit în fișierul header IOMANIP.H. Următorul program, SETBASE.CPP, folosește manipulatorul *setbase* pentru a obține același rezultat ca și programul anterior:



```
#include <iostream.h>
#include <iomanip.h>

void main(void){
    int number = 1001;

    cout << "Decimal: " << number << "\tHexadecimal: " <<
        setbase(16) << number << endl;
    cout << "Decimal: " << setbase(10) << number << "\tOctal: " <<
        setbase(8) << number << endl;
}
```

**Observație:** Dacă compilatorul pe care îl folosiți nu include manipulatorul **setbase**, puteți folosi manipulatorul **setiosflags**. De exemplu, pentru a selecta baza hexazecimală, folosiți **setiosflags (ios::hex)**.

### CONTROLUL LĂȚIMII ZONEI DE ALINIERE

Câteva din programele anterioare au folosit spații în interiorul șirurilor de caractere pentru a formata afișarea acestora. Pe lângă această metodă „brutală” de spațiere, programele pot folosi manipulatorul **setw** care specifică numărul minim de poziții ale zonei de alinere. Dacă o valoare necesită mai multe caractere decât numărul specificat în **setw**, atunci pentru afișarea ei vor fi folosite atâtea caractere câte sunt necesare. Dacă o valoare necesită mai puține caractere, atunci spații suplimentare vor fi inserate înaintea valorii afișate. Spre deosebire de manipulatorii **dec**, **oct**, **hex** și **setbase** prezentați mai înainte, manipulatorul **setw** are efect numai asupra valorii următoare. Cu alte cuvinte, alinierea stabilită prin **setw** nu este permanentă. Următorul program, SETW.CPP, ilustrează folosirea manipulatorului **setw**:

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
{
    int i;

    for (i = 1; i < 5; i++)
        cout << setw(i) << 1 << setw(i) << 12 << setw(i) << 123 << endl;
}
```

Deoarece lățimea zonei de aliniere selectată prin **setw** influențează numai următoarea valoare afișată, programul folosește manipulatorul de câteva ori în aceeași instrucțiune. La compilarea și execuția acestui program, pe ecran vor fi afișate următoarele:

```
C:\> SETW <ENTER>
112123      setw(1)
112123      setw(2)
```

```
1 12123      setw(3)
1 12 123     setw(4)
```

După cum se poate observa, dacă o valoare are mai multe caractere decât cele selectate de **setw**, ea va fi afișată cu toate caracterele necesare. În cazul primei linii de ieșire, toate cele trei valori conțin cel puțin o cifră. În a doua linie sunt alocate două caractere pentru fiecare valoare, în a treia linie - trei, iar în a patra linie - 4 caractere. Se observă că în ultima linie, valoarea 1 este precedată de 3 spații, valoarea 12 - de două spații, iar valoarea 123 - de un spațiu.

### STABILIREA CARACTERULUI DE UMPLERE

Așa cum ați învățat, manipulatorul **setw** permite specificarea lățimii minime a zonei de aliniere. În mod prestabilit, stream-urile I/O folosesc spațiile pentru obținerea unei spațieri corecte între caractere. Cu ajutorul manipulatorului **setfill** se pot specifica și alte caractere de completare a zonelor dintre valori. De exemplu, următoarea instrucțiune dirijează stream-ul **cout** pentru a folosi punctul drept caracter de completare:

```
cout << setfill('.');
```

Când se specifică un caracter prin **setfill**, acel caracter rămâne în vigoare pentru toate afișările ulterioare, până când se selectează un alt caracter. Următorul program, SETFILL.CPP, schimbă programul precedent SETW.CPP pentru a folosi caracterul '.' drept caracter de umplere:

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
{
    int i;

    cout << setfill('.');

    for (i = 1; i < 5; i++)
        cout << setw(i) << 1 << setw(i) << 12 << setw(i) << 123 << endl;
}
```

Prototipul pentru manipulatorul **setfill** se găsește în fișierul antet IOMANIP.H. La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> SETFILL <ENTER>
.112123
..112123
...1.12123
```

O utilizare frecventă a manipulatorului *setfill* este la crearea unui meniu sau a unei table de cuprins, după cum se vede în continuare:

Cuprins	
Capitolul 1. Inițierea în intrări/ieșiri cu tastatura	1
Capitolul 2. Inițierea în clase	22
Capitolul 3. Inițierea în fișiere	45
Capitolul 4. Inițierea în clase template	67
Capitolul 5. Inițierea în moștenire	99

Următorul program, TABLE.CPP, ilustrează modul în care manipulatorul *setfill* creează tabela precedentă:

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
{
    cout << setfill('.');
    cout << "Table of Contents" << endl;
    cout << "Chapter 1 Getting Up to Speed with Keyboard I/O" <<
        setw(18) << 1 << endl;
    cout << "Chapter 2 Getting Up to Speed with Classes" <<
        setw(34) << 22 << endl;
    cout << "Chapter 3 Getting Up to Speed with Files" <<
        setw(36) << 45 << endl;
    cout << "Chapter 4 Getting Up to Speed with Templates" <<
        setw(32) << 67 << endl;
    cout << "Chapter 5 Getting Up to Speed with Inheritance" <<
        setw(30) << 99 << endl;
}
```

### CONTROLUL AFIȘĂRII VALORILOR ÎN VIRGULĂ MOBILĂ<sup>1</sup>

Când se lucrează cu numere zecimale, ca de exemplu 3,14159, se poate controla numărul cifrelor afișate prin folosirea manipulatorului *setprecision*. Să presupunem că dorim să tipărim rezultatul expresiei 22,0/7,0. Pentru a controla numărul de zecimale afișate se folosește manipulatorul *setprecision*, ca în programul SETPREC.CPP:

```
#include <iostream.h>
#include <iomanip.h>
```

```
void main(void)
```

<sup>1</sup> După cum se știe, sistemul zecimal anglo-saxon folosește virgula în locul punctului și invers. – N.R.

```
{
    int i;

    for (i = 0; i < 10; i++)
        cout << setprecision(i) << 22.0 / 7.0 << endl;
}
```

Așa cum se vede, programul ciclează de la 0 la 10, folosind *setprecision* pentru controlul numărului de cifre afișate. Prototipul manipulatorului *setprecision* este definit în fișierul header IOMANIP.H. La compilare și execuția acestui program, pe ecran vor fi afișate următoarele:

```
C:\> SETPREC <ENTER>
3.142857
3.1
3.14
3.143
3.1429
3.14286
3.142857
3.1428571
3.14285714
```

**Observații:** Tot în acest capitol veți învăța să selectați formatul fix de afișare a numerelor zecimale folosind indicatorul *ios::fixed*. La folosirea unui format fix, manipulatorul *setprecision* controlează numărul de zecimale afișate, și nu numărul total de cifre.

### GOLIREA BUFFER-ULUI

Așa cum ați învățat, stream-urile de ieșire *cout* și *clog* folosesc ieșirea prin buffer (memoria tampon), adică datele sunt memorate într-o locație de memorie și apoi transferate simultan, în întregime, pentru o mai mare eficiență. Tot în acest capitol, veți examina în detaliu ieșirea prin buffer și ieșirea normală (fără buffer). Veți învăța că, atunci când realizați ieșirea prin buffer, sunt momente când doriți ca programul să transfere imediat datele (să le mute din locația buffer-ului) pe ecran. Pentru a obține ieșirea în acest mod, se poate folosi manipulatorul *flush*. Programele prezentate în secțiunea respectivă vor explica în detaliu noțiunile legate de manipulatorul *flush* (fără buffer).

### IGNORAREA SPAȚIILOR ALBE CE PRECED DATELE DE INTRARE

După cum știți, stream-ul de intrare *cin* folosește spații albe pentru a separa câmpurile de intrare. Când realizați operații I/O, există cazuri când *cin* trebuie să ignore spațiile anterioare primului câmp de informații. Pentru aceasta se poate folosi manipulatorul *ws*, ca mai jos:

```
cin >> ws >> camp;
```

## Succes cu C++

De exemplu, următorul program, SKIL\_WS.CPP, folosește manipulatorul *ws* pentru ignorarea spațiilor albe plasate înaintea textului tipărit:

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
{
    char text[128];

    cout << "Type in a word preceded by one or more blanks" << endl;
    cin >> ws >> text;
    cout << "You typed >" << text << "<" << endl;
}
```

## CONTROLUL INDICATORILOR STREAM-ULUI I/O

Așa cum ați învățat, o parte din manipulatorii stream-ului I/O rămân activi și după încheierea unei operații de intrare/ieșire. De exemplu, manipulatorii *dec*, *hex* și *oct* selectează baza de afișare a valorilor numerice. Pentru a implementa acești manipulatori, stream-urile I/O au un câmp indicator (flag field) care specifică valorile curente ale parametrilor. Un fișier antet definește aceste valori ca fiind de tip enumerativ, după cum se vede mai jos:

```
enum {
    skipws = 0x0001,    // Skip white space during input
    left = 0x0002,      // Left justify output
    right = 0x0004,     // Right justify output
    internal = 0x0008,  // Pad after sign or base indicator
    dec = 0x0010,       // Decimal base
    oct = 0x0020,       // Octal base
    hex = 0x0040,       // Hexadecimal base
    showbase = 0x0080,  // Use base indicator on output
    showpoint = 0x0100, // Force decimal point for floating point
    uppercase = 0x0200, // Uppercase hex output
    showpos = 0x0400,   // Add '+' to positive integers
    scientific = 0x0800, // Use 3.1415E2 floating notation
    fixed = 0x1000,     // Use 314.45 floating notation
    unitbuf = 0x2000,   // Flush all streams after insertion
    stdio = 0x4000,     // Flush stdout, stderr after insertion
}
```

Prin folosirea manipulatorilor *setiosflags*, programele pot controla o bună parte dintre acești indicatori. De exemplu, următorul program, IOSFLAGS.CPP, folosește indicatorii *dec*, *oct*, *hex* și *showbase* pentru a afișa un număr în zecimal, octal și hexazecimal:

## I: Tastatura și ecranul

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
{
    cout << setiosflags(ios::showbase);

    cout << setiosflags(ios::dec);
    cout << "10 in decimal is " << 10 << endl;

    cout << setiosflags(ios::oct);
    cout << "10 in octal is " << 10 << endl;

    cout << setiosflags(ios::hex);
    cout << "10 in hex is " << 10 << endl;
}
```

La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> IOSFLAGS <ENTER>
10 in decimal is 10
10 in octal is 012
10 in hex is 0xa
```

În acest caz, indicatorul *ios::showbase* dirijează stream-ul *cout* să prefixeze valorile octale cu 0, iar cele hexazecimale cu 0x. În unele cazuri, poate fi nevoie de setarea mai multor indicatori simultan. Aceasta se poate realiza printr-o operație OR (SAU) între indicatorii doriți, cum se vede în continuare:

```
cout << setiosflags (ios::showbase / ios::oct);
```

## AFIȘAREA VALORILOR HEXAZECIMALE CU MAJUSCULE

După cum știți, manipulatorul *hex* permite afișarea unei valori în hexazecimal. În mod prestabilit, stream-urile I/O afișează valorile hexazecimale cu litere mici. Pentru ca acestea să fie afișate cu majuscule, se poate folosi indicatorul *ios::uppercase*. Programul următor, UPPERCASE.CPP, prezintă modalitatea de utilizare a indicatorului menționat.

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
{
    cout << "Lowercase: " << hex << 255 << " " << 10 << endl;
    cout << setiosflags(ios::uppercase);
    cout << "Uppercase: " << 255 << " " << 10 << endl;
}
```

La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> UPPERHEX <ENTER>
Lowercase: ff a
Uppercase: FF A
```

### ALINIAREA REZULTATELOR LA STÂNGA SAU LA DREAPTA

În mod prestabilit, stream-urile I/O afișează ieșirea aliniată la stânga. Prin folosirea indicatorilor `ios::left` și `ios::right`, programele pot controla alinierea ieșirii. De exemplu, următorul program, `LEFTRITE.CPP`, folosește indicatorii menționați pentru a alinia ieșirea atât la stânga, cât și la dreapta:

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
{
    cout << setiosflags(ios::right);
    cout << setw(5) << 1 << setw(5) << 2 << setw(5) << 3 << endl;
    cout << setiosflags(ios::left);
    cout << setw(5) << 1 << setw(5) << 2 << setw(5) << 3 << endl;
}
```

După compilarea și execuția programului, ecranul va arăta astfel:

```
C:\> LEFTRITE <ENTER>
    1    2    3
1    2    3
```

După cum s-a văzut, programul anterior ar fi putut conține instrucțiuni combinate, astfel:

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
{
    cout << setiosflags(ios::right) << setw(5) << 1 << setw(5) <<
        2 << setw(5) << 3 << endl;
    cout << setiosflags(ios::left) << setw(5) << 1 << setw(5) <<
        2 << setw(5) << 3 << endl;
}
```

### CONTROLUL AFIȘĂRII CU PUNCT FIX ȘI CU EXPONENT

Valorile reale pot fi afișate atât în format zecimal fix, ca de exemplu 123.456, cât și în format cu exponent (notație științifică), cum ar fi 1.23456e2. Indicatorii `ios::fixed` și `ios::scientific` permit controlul afișării numerelor reale. Următorul program, `FIXFLOAT.CPP`, ilustrează folosirea acestor doi indicatori de formatare:

```
#include <iostream.h>
#include <iomanip.h>
```

```
void main(void)
{
    cout << setiosflags(ios::fixed) << 123.45 << endl;
    cout << 12345.6789 << endl;
    cout << resetiosflags(ios::fixed);
    cout << setiosflags(ios::scientific) << 123.45 << endl;
    cout << 12345.6789 << endl;
}
```

**Observație:** Programul folosește instrucțiunea `resetios flags (ios::fixed)` pentru preîntâmpinarea erorilor ce pot apărea la rularea programelor din această carte cu unele compilatoare. Testați acest program folosind compilatorul pe care îl aveți la dispoziție și decideți dacă puteți elimina această instrucțiune.

La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> FIXFLOAT <ENTER>
123.450000
12345.678900
1.234500e+02
1.234568e+04
```

După cum se observă, formatul `ios::fixed` dirijează stream-ul I/O pentru afișarea punctului zecimal în poziția sa actuală, în timp ce indicatorul `ios::scientific` dirijează stream-ul I/O să folosească formatul exponențial.

### FORȚAREA AFIȘĂRII PUNCTULUI ZECIMAL

Când programele afișează rezultatele unor operații în virgulă mobilă, nu întotdeauna punctul zecimal este tipărit, în special când rezultatele sunt exacte (fără parte zecimală). Să considerăm, de exemplu, următoarea instrucțiune:

```
cout << 10.0/5 << endl;
```

Deoarece rezultatul împărțirii este exact, stream-ul I/O poate tipări valoarea 2, în loc de 2.0. Dacă se dorește și tipărirea punctului zecimal, se poate folosi indicatorul `ios::showpoint`. Următorul program, `SHOWPOIN.CPP` ilustrează folosirea acestui indicator pentru forțarea afișării punctului zecimal.

```
#include <iostream.h>
#include <iomanip.h>
```

```
void main(void)
{
    cout << 10.0 / 5 << endl;
    cout << setiosflags(ios::showpoint) << 10.0 / 5 << endl;
}
```

La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> SHOWPOIN <ENTER>
2
2.00000
```

## FORȚAREA AFIȘĂRII SEMNULUI UNEI VALORI

În mod prestabilit, stream-urile I/O afișează numai semnul valorilor negative. În funcție de scopul programului, uneori este necesară afișarea semnului plus în fața valorilor pozitive. În astfel de situații, programele pot folosi indicatorul `ios::showpos`. De exemplu, următorul program, `SHOWPOS.CPP`, ilustrează folosirea acestui indicator.

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
{
    cout << -10 << " " << -5 << " " << 0 << " " << 5 << " " <<
        10 << endl;
    cout << setiosflags(ios::showpos);
    cout << -10 << " " << -5 << " " << 0 << " " << 5 << " " <<
        10 << endl;
}
```

După compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> SHOWPOS <ENTER>
-10 -5 0 5 10
-10 -5 0 +5 +10
```

## REFACEREA INDICATORILOR STREAM-URILOR I/O

În acest paragraf ați învățat cum se selectează anumiți indicatori prin folosirea manipulatorului `setiosflag`. Dacă programele execută o mare cantitate de stream I/O, uneori este necesară activarea/dezactivarea indicatorilor. Pentru a readuce un indicator la valoarea sa prestabilită, programele pot folosi manipulatorul `resetiosflags`. De exemplu, următorul program, `RESETIO.CPP`, va afișa diferite valori în hexazecimal, precedate de specificatorul bazei hexazecimale `0x`. Programul folosește apoi manipulatorul `resetiosflags` pentru a invalida afișarea specificatorului bazei:

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
```

```
{
    cout << setiosflags(ios::showbase);
    cout << hex << 255 << " " << 10 << endl;
    cout << resetiosflags(ios::showbase);
    cout << 255 << " " << 10 << endl;
}
```

La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> RESETIO <ENTER>
0xff 0xa
ff a
```



## FOLOSIREA INDICATORILOR I/O

Manipulatorii stream-urilor permit programelor să controleze diferite setări I/O, prin păstrarea în stream-urile I/O a unei colecții de biți indicatori (flag bits). Prin folosirea manipulatorilor `setiosflags` și `resetiosflags`, programele pot controla direct starea indicatorilor. De exemplu, următoarea instrucțiune dirijează stream-ul `cout` să afișeze informațiile aliniate la dreapta:

```
cout << setiosflags (ios::right);
```

Pentru a înțelege mai bine stările indicatorilor I/O, se recomandă examinarea fișierului `antet` folosit de compilator pentru definițiile legate de stream-uri (cum sunt `IOSTREAM.H`, `ISTREAM.H`, `OSTREAM.H`). În Capitolul 12 veți învăța să creați proprii dumneavoastră manipulatori, pentru a simplifica folosirea acestor indicatori în programe.

## FACILITĂȚI OFERITE DE FUNCȚIILE MEMBRU ALE STREAM-ULUI DE INTRARE

Programele C++ mai simple folosesc pentru operația de intrare fluxul `cin` și operatorul de inserție. Pe măsură ce programele devin mai complexe, va fi necesar un control mai precis asupra operațiilor de intrare, decât cel oferit de operatorul de inserție. În astfel de situații, programele pot folosi funcțiile membru de intrare ale stream-ului I/O, discutate în acest capitol.

## DETERMINAREA NUMĂRULUI DE CARACTERE EXTRASE

În momentul extragerii caracterelor dintr-un stream de intrare folosind operatorul de extracție, uneori este necesară cunoașterea numărului de caractere extrase. În acest scop, programele pot folosi funcția membru `gcount`. De exemplu, programul următor, `GCOUNT.CPP`, vă cere să introduceți o linie de text.

Apoi, programul citește textul folosind funcția `cin.getline`, iar cu ajutorul funcției membru `gcount` determină numărul de caractere citite:

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
{
    char text[64];

    cout << "Type in a line of text and press Enter" << endl;

    cin.getline(text, sizeof(text));

    cout << "You typed: " << text << " " << cin.gcount() <<
        " characters" << endl;
}
```

**Observație:** La contorizarea caracterelor unui *string* (șir de caractere), funcția `gcount` include și caracterul CR (carriage return - retur de car) - adică, dacă introduceți șirul 123, valoarea lui `gcount` va fi 4.

Compilați și executați acest program. Experimentați funcția `gcount` introducând șiruri cu număr diferit de caractere. Aveți în vedere că `gcount` include și caracterul de sfârșit de linie CR.

### \*CITIREA UNEI LINII DE LA TASTATURĂ SAU DE LA STDIN

Când programele realizează operații de intrare folosind stream-ul de intrare `cin`, acesta separă valorile prin spații albe (spații simple, tabulatori sau caractere sfârșit de linie). În funcție de valorile pe care programele trebuie să le introducă, poate fi necesară citirea unei linii întregi de text, ca apoi părți ale acesteia să fie prelucrate. Pentru citirea unei linii de text, programele pot folosi funcția membru `getline`. Următorul program, `GETLINE.CPP`, folosește funcția membru `getline` pentru citirea unei linii de text de la tastatură.

```
#include <iostream.h>

void main(void)
{
    char line[128];

    cout << "Type in a line of text and press Enter" << endl;
    cin.getline(line, sizeof(line));
    cout << "You typed: " << line << endl;
}
```

În mod prestabilit, funcția membru `getline` citește caracterele până la întâlnirea caracterului sfârșit de linie sau până la un număr specificat de caractere. În funcție de cerințele de intrare ale programului, uneori este necesară oprirea ope-

rației de intrare la întâlnirea unui anumit caracter. Pentru a termina operația de intrare când `getline` întâlnește litera X, se poate apela funcția membru astfel:

```
cin.getline (line, sizeof (line), 'X');
```

La stoparea operațiilor I/O folosind un anumit caracter, este important să știm că următoarea operație de intrare va continua la întâlnirea caracterului imediat următor caracterului de stopare. De exemplu, următorul program `STOPON_X.CPP`, încheie prima operație de intrare pe caracterul X. Programul realizează apoi a doua operație de intrare.

```
#include <iostream.h>

void main(void)
{
    char line[128];

    cout << "Type in a line of text and press Enter" << endl;
    cin.getline(line, sizeof(line), 'X');
    cout << "First line: " << line << endl;
    cin.getline(line, sizeof(line));
    cout << "Second line: " << line;
}
```

Compilați și executați acest program. Dați la intrare o serie de cuvinte separate de caracterul X. Programul va afișa textul ce precede caracterul X pe prima linie și textul ce urmează după X pe a doua linie.

**Observație:** Caracterul X în acest program este dependent de tipul literei (case-sensitive). Litera mică 'x' nu va determina oprirea intrării la primul apel al funcției `cin.getline`. De asemenea, până la introducerea caracterului X, puteți tasta ENTER ori de câte ori doriți, fără a opri operația de intrare la primul apel al lui `cin.getline`; aceasta se va încheia la prima tastare ENTER după introducerea caracterului X.



### FOLOSIREA FUNCȚIILOR MEMBRU ALE STREAM-ULUI CIN

Așa cum s-a spus, `cin` este un obiect de tipul `istream`. De aceea `cin` suportă mai multe funcții. Următoarea instrucțiune, de exemplu, folosește funcția membru `getline` pentru a citi o linie de text de la tastatură:

```
cin.getline (line, sizeof (line));
```

Examinând definiția stream-ului `cin` din fișierul `IOSTREAM.H`, puteți determina rapid funcțiile membru disponibile prin vizualizarea prototipurilor acestora.

## REALIZAREA OPERAȚIILOR DE INTRARE CARACTER CU CARACTER

În funcție de cerințele de intrare ale unui program, uneori sunt necesare operații de introducere a câte unui singur caracter. În astfel de situații se poate folosi funcția membru *get*. De exemplu, programul următor, YES\_NO.CPP, invită utilizatorul să introducă un răspuns prin una din alternativele Y sau N. Programul folosește funcția membru *get* până când întâlnește caracterul Y sau N:

```
#include <iostream.h>
#include <ctype.h>

void main(void)
{
    char letter;

    cout << "Type a Y or N: ";

    do {
        letter = cin.get();           // Read a character
        letter = toupper(letter);     // Convert to uppercase
    } while ((letter != 'Y') && (letter != 'N'));

    cout << endl << "You typed: " << letter << endl;
}
```

Așa cum se poate vedea, programul ciclează până la întâlnirea unui caracter Y sau N. Pentru simplificarea testului, programul convertește fiecare literă citită în majusculă.

Rețineți că operațiile de intrare ce folosesc stream-ul I/O *cin* nu se pot executa numai de la tastatură. Următorul program, TO\_UPPER.CPP, folosește funcția membru *get* pentru a citi câte un caracter de la intrarea redirecționată, convertind fiecare literă în majusculă, până când este întâlnit sfârșitul de fișier:

```
#include <iostream.h>
#include <ctype.h>

void main(void)
{
    char letter;

    while (!cin.eof())
    {
        letter = cin.get();
        letter = toupper(letter);
        cout << letter;
    }
}
```

După cum se vede, programul folosește funcția membru *eof* pentru detectarea sfârșitului de fișier (sau al intrării redirecționate). Tot în acest capitol vom examina funcția *eof* în detaliu.

Următoarea comandă folosește comanda TO\_UPPER pentru afișarea conținutului fișierului TO\_UPPER.CPP cu litere majuscule:

```
C:\> TO_UPPER < TO_UPPER.CPP <Enter>
```

## ANTICIPAREA URMĂTORULUI CARACTER LA INTRARE

În funcție de cerințele de intrare ale unui program, uneori este necesară citirea unor caractere până la un anumit caracter, cu excepția acestuia. De exemplu, să presupunem că un program citește un fișier ce conține nume urmate de numere de telefon, ca mai jos:

```
Joe Smith 555-1212
John Davis 222-2323
Betty Lou Johnson 333-3343
```

În acest caz, programul ar putea atribui caracterele ce preced numărul de telefon unei variabile de tip șir. Pentru a citi caracterele până la un anumit caracter, se poate folosi funcția membru *cin.peek*. Funcția citește un caracter din buffer-ul de intrare, fără a-l scoate din buffer. Următorul program, TO\_DIGIT.CPP, vă cere să introduceți cuvinte, urmate de un număr. Programul folosește funcția membru *peek* pentru a localiza prima cifră, atribuind caracterele care preced numărul unui șir de caractere denumit *line*:

```
#include <iostream.h>
#include <ctype.h>

void main(void)
{
    char letter, string[128];

    int i = 0, done = 0;

    cout << "Type in a string terminated by a number" << endl;

    do {
        letter = cin.peek();

        if (!isdigit(letter))
            string[i++] = cin.get();
        else
            done = 1;
    } while ((!done) && (i < sizeof(string)));

    string[i] = NULL;

    cout << "String input: " << string << endl;
}
```



**Observație:** Majoritatea programelor ce folosesc funcția membru **peek** pot fi rescrise pentru a elimina necesitatea programelor de a folosi funcția, îmbunătățindu-le astfel claritatea. Dacă folosiți în programe funcția **peek**, trebuie să aveți în vedere și modalități de a rescrie programele.

#### READUCEREA UNUI CARACTER ÎN BUFFER-UL DE INTRARE

Programul anterior folosea funcția membru *peek* a stream-ului de intrare pentru a determina următorul caracter din buffer-ul de intrare. În funcție de operațiile efectuate de un program, uneori poate fi necesară plasarea unui caracter în buffer-ul de intrare sau plasarea în buffer a altui caracter decât cel citit. În astfel de situații, programele pot folosi funcția membru *putback*. Următorul program, **PUTBACK.CPP**, citește caractere din buffer-ul de intrare, afișând numărul de caractere citite. Programul contorizează numai caracterele majuscule. La întâlnirea unei litere mici, programul plasează înapoi în buffer majuscula echivalentă:

```
#include <iostream.h>
#include <ctype.h>

void main(void)
{
    int count[26];
    char i;

    char letter;

    for (i = 0; i < 26; i++)
        count[i] = 0;

    cout << "Type in a string and press Enter" << endl;

    do {
        letter = cin.get();

        if ((letter >= 'a') && (letter <= 'z'))
            cin.putback(toupper(letter));
        else if ((letter >= 'A') && (letter <= 'Z'))
            count[letter - 'A']++;
    } while (letter != '\n');

    for (i = 0; i < 26; i++)
    {
        cout.put((char)('A' + i));
        cout << " " << count[i] << endl;
    }
}
```

După cum se observă, dacă programul întâlnește o literă mică, el folosește funcția membru *putback* pentru a plasa litera majusculă echivalentă în buffer-ul de intrare. În acest mod, programul va contoriza numai literele majuscule.

**Observație:** Majoritatea programelor care folosesc funcția membru *putback* pot fi rescrise fără a folosi această funcție, îmbunătățindu-se astfel claritatea acestora. Dacă folosiți în programe funcția *putback* e bine să vă gândiți la modalități de a rescrie acele programe.

#### DETECTAREA SFÂRȘITULUI DE FIȘIER

Capitolul 3 examinează în detaliu operațiile cu fișiere C++. Veți afla atunci că multe programe citesc fișierele de la început până la sfârșit, folosind funcția membru *eof* de detectare a sfârșitului de fișier. Când scrieți programe ce folosesc redirectarea I/O, puteți folosi funcția *eof* pentru depistarea sfârșitului intrării redirectate. De exemplu, următorul program, **LINE.CPP**, citește dintr-o intrare redirectată și apoi afișează numărul de linii citite:

```
#include <iostream.h>

void main(void)
{
    char line[256];
    long count = 0;

    while (!cin.eof())
    {
        cin.getline(line, sizeof(line));
        count++;
    }

    cout << "Lines read: " << count << endl;
}
```

După compilarea acestui program, îi puteți redireja intrarea, astfel:

```
C:\> LINECNT < LINECNT.CPP <ENTER>
Lines read: 15
```

După cum se vede, acest program, ca și altele prezentate în acest capitol, ciclează până la întâlnirea sfârșitului de fișier.

#### IGNORAREA CARACTERELOR DIN STREAM-UL DE INTRARE

La realizarea operațiilor de I/O, pot exista situații când se dorește ignorarea caracterelor din stream-ul de intrare. De exemplu, să presupunem că un program trebuie să citească următorul fișier și are nevoie numai de numerele de asigurări sociale:

Smith, John	111-22-3333
Lewis, Bill	222-33-4444
Jones, Mary	333-44-5555

Folosind funcția membru *ignore*, se pot ignora caractere din stream-ul de intrare. De exemplu, următoarea instrucțiune dirijează *cin* să ignore următoarele 10 caractere de la intrare:

```
cin.ignore(10);
```

Programele pot, de asemenea, să dirijeze stream-ul I/O pentru a ignora caracterele până la întâlnirea unui anumit caracter. De exemplu, următoarea instrucțiune determină *cin* să ignore primele 10 caractere sau caracterele până la litera A inclusiv:

```
cin.ignore(10, 'A');
```

În acest caz, *cin* va ignora cel mult 10 caractere. Cu alte cuvinte, dacă litera A apare în primele 10 caractere, *cin* nu va mai ignora caracterele ce apar după litera A.

### CITIREA ȘI SALVAREA DIFERITELOR VALORI ALE PARAMETRILOR I/O

Așa cum ați învățat, stream-urile I/O furnizează câțiva manipulatori și funcții ce permit controlul unor parametri I/O. De exemplu, manipulatorul *setw* permite definirea lățimii unui câmp de afișare. Când programele folosesc acești manipulatori pentru a controla diferite valori ale unor parametri, poate fi necesară determinarea valorilor indicatorilor curenți, astfel încât programele să poată fi reconstituite mai târziu. În astfel de situații, se pot folosi funcțiile din tabela 1.3. pentru citirea valorilor curente.

Funcția membru	Rezultatul returnat
flags	valoarea indicatorilor curenți
width	lățimea câmpului curent de afișare
fill	caracterul de completare
precision	valoarea preciziei curente

**Tabela 1.3.** Funcțiile membru ale stream-ului I/O ce returnează valori ale parametrilor de stare

Următorul program, SHOWINFO.CPP, folosește aceste funcții pentru a afișa valorile prestabilite ale parametrilor unui stream:

```
#include <iostream.h>

void main(void)
{
    long flags;

    flags = cout.flags();
    cout << "Default flags: " << hex << flags << endl;
    cout << "Default width: " << dec << cout.width() << endl;
```

```
    cout << "Default fill: "<< cout.fill() << "<" << endl;
    cout << "Default precision: " << cout.precision() << endl;
}
```

La compilarea și execuția acestui program, pe ecran se va afișa:

```
C:\> SHOWINFO <ENTER>
Default flags: 2001
Default width: 0
Default fill: > <
Default precision: 6
```



### SĂ ÎNȚEGEM PARAMETRII STREAM-ULUI DE IEȘIRE

C++ dispune de o serie de funcții membru ce permit controlul spațierii și formatării unui stream. Când folosiți aceste funcții pentru a schimba o valoare de parametru, schimbarea rămâne activă până la terminarea programului sau efectuarea altei schimbări. Dacă examinați fișierul antet folosit de compilator pentru definițiile de stream-uri, veți afla că aceste funcții corespund variabilelor membru ale unei clase de stream I/O. La invocarea funcției, valoarea variabilei membru corespunzătoare este schimbată. Ulterior, la realizarea unei operații I/O cu acel stream, operația va folosi valoarea stabilită a variabilei membru pentru controlul intrării sau ieșirii.

### FACILITĂȚI OFERITE DE FUNCȚIILE MEMBRU ALE STREAM-ULUI DE IEȘIRE

Așa cum ați învățat, stream-urile de intrare furnizează câteva funcții membru pentru realizarea unor operații I/O specifice. Acest paragraf examinează funcții membru ale stream-ului de ieșire cu aceeași finalitate. De exemplu, următorul program, PUTALPHA.CPP, folosește funcția membru *put* pentru a afișa pe ecran literele alfabetului, una câte una:

```
#include <iostream.h>

void main(void)
{
    char letter;

    for (letter = 'A'; letter <= 'Z'; letter++)
        cout.put(letter);
}
```


La compilarea și execuția programului, pe ecran va apărea:

```
C:\> PUTALPHA <ENTER>
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Poate vă întrebați de ce în programul precedent s-a folosit funcția membru *put*, în loc de *cout*, ca mai jos:

```
for (letter = 'A'; letter <= 'Z'; letter++)
    cout << letter;
```

Dacă în program variabila *letter* este declarată de tip *char*, se pot folosi ambele tehnici pentru afișarea literelor. Însă dacă variabila *letter* este de tip *int*, atunci prin a doua metodă pe ecran se vor afișa numerele între 65 și 90, în loc de literele A până la Z.



**FOLOSIREA FUNCȚIILOR MEMBRU  
ALE STREAM-ULUI COUT**

Așa cum s-a văzut, *cout* este un obiect de tip *ostream*. Ca atare, *cout* suportă mai multe funcții membru. Următoarea instrucțiune, de exemplu, folosește funcția membru *put* pentru a scrie un caracter la dispozitivul standard de ieșire:

```
cout.put(7); // semnal sonor prin difuzorul incorporat
```

Prin examinarea definiției clasei *cout* din fișierul *IOSTREAM.H*, se pot găsi rapid funcțiile membru disponibile, prin consultarea listei de prototipuri.

## TESTAREA REUȘITEI OPERAȚIILOR I/O

Fiecare din programele anterioare au realizat operații de I/O, presupunând că acele operații s-au derulat cu succes. Când programele scriu mesaje simple pe ecran sau citesc una-două linii de la tastatură, în mod normal putem presupune că operațiile se încheie cu succes. Însă, pe măsură ce programele devin mai complexe, ar trebui verificată finalitatea operațiilor, folosind funcțiile membru *good*, *bad*, *fail* și *rdstate*. Fiecare din aceste funcții membru examinează biții de stare din indicatorul de stare al stream-ului I/O, pentru a determina dacă a avut loc o eroare. În funcție de compilatorul folosit, valoarea și semnificația biților de stare poate fi ușor diferită. Totuși, dacă examinați fișierul antet al definițiilor legate de stream-uri, veți găsi o definiție de tip enumerativ a acestor biți, ca mai jos:

```
enum io_state {
    goodbit = 0x00 // Set if I/O operations have been successful
    eofbit = 0x01  // Set if currently at the end of the file
    failbit = 0x02  // Set if last I/O operation failed
    badbit = 0x04   // Set if an invalid operation was attempted
    hardfail = 0x80 // Set if an unrecoverable error occurred
};
```

Pentru a vedea dacă a avut loc o eroare I/O, programele pot testa diferiți biți de stare. De exemplu, următorul program, *TEST\_IO.CPP*, folosește funcția membru *fail* pentru a determina dacă operațiile I/O s-au încheiat cu succes:

```
#include <iostream.h>
#include <stdlib.h>
#include <ctype.h>

void main(void)
{
    char letter;

    while (!cin.eof())
    {
        letter = cin.get();
        if (cin.fail())
        {
            cerr << "Error reading input" << endl;
            exit(1);
        }

        cout.put((char)toupper(letter));
        if (cout.fail())
        {
            cerr << "Error writing output" << endl;
            exit(1);
        }
    }
}
```

Să presupunem, de exemplu, că programul precedent va fi apelat astfel:

```
C:\> TEST_IO < FILENAME.CPP > A:UPPER.CPP <ENTER>
```

În acest caz, *TEST\_IO* va citi conținutul fișierului *FILENAME.CPP*, scriind majuscule echivalente fiecărui caracter pe fișierul *UPPER.CPP* de pe unitatea A. Dacă apare o eroare când *TEST\_IO* citește fișierul de intrare sau scrie fișierul de ieșire (de pildă, spațiu insuficient pe disc), programul va afișa un mesaj de eroare în stream-ul fișierului standard de erori și se va termina. Deoarece programele pot testa corectitudinea operațiilor I/O în mod obișnuit, stream-urile I/O definesc semnul de exclamare ca un operator ce detectează o operație I/O nereușită. Astfel, următoarele instrucțiuni *if* sunt identice din punct de vedere funcțional:

```
if (cin.fail())           if (!cin)
```


În cazul când un stream I/O întâlnește o eroare, bitul de eroare rămâne setat până când se șterge indicatorul prin funcția membru *clear*, cum se arată mai jos:

```
if (cout.fail())
{
    cerr << "Display some error message" << endl;
    cout.clear();
}
```

În funcție de modul în care un program manipulează erorile I/O, se poate cere examinarea indicatorilor de stare în vederea determinării tipului de eroare apărută. Pentru aceasta se poate folosi funcția membru *rdstate*, ca mai jos:

```
io_state = cin.rdstate();
```

Pentru a înțelege mai bine definițiile acestor funcții membru, puteți examina fișierul antet folosit de compilator pentru definițiile legate de stream-urile I/O.



### SĂ ÎNȚELEM STAREA UNUI STREAM I/O

La realizarea operațiilor cu stream-urile I/O, se poate testa în programe finalizarea cu succes a unei operații folosind funcțiile membru ale clasei I/O. Dacă examinați fișierul antet pentru definițiile legate de stream-uri, veți afla că stream-urile I/O conțin o variabilă membru de stare ai cărei biți corespund stării curente a stream-ului. Analizând biții acestei variabile, programele pot determina încheierea cu succes a operației I/O anterioare. Majoritatea programelor, totuși, folosesc funcții membru încorporate pentru a realiza aceste teste.

## INTRĂRI-IEȘIRI CU ȘI FĂRĂ BUFFER

Stream-urile I/O *cout* și *clog* folosesc ieșirea prin buffer, adică ieșirea nu este afișată decât când buffer-ul se umple, programul se încheie, buffer-ul este golit intenționat, sau, în cazul stream-ului *cout*, când programul citește din stream-ul *cin*. Următorul program, *BUFFERED.CPP*, scrie ieșirea la *cout* și așteaptă apoi 3 secunde până la terminare. Când programul ia sfârșit, buffer-ul de ieșire este afișat, ca în următorul exemplu:

```
#include <iostream.h>
#include <time.h>

void main(void)
{
    time_t start_time, current_time;

    cout << "Hello C++ world!" << endl;
    time(&start_time);

    do {
        time(&current_time);
    } while ((current_time - start_time) < 3);
}
```

Programul folosește funcția *time* pentru determinarea timpului curent și introduce o întârziere de 3 secunde.

## GOLIREA UNUI BUFFER DE IEȘIRE

În funcție de operațiile efectuate de program apar situații când buffer-ul de ieșire trebuie golit înainte de a se umple, înainte de încheierea programului, sau înainte de a realiza o operație de intrare. În astfel de situații programele pot folosi funcția membru *flush*. De exemplu, următorul program, *FLUSH.CPP*, scrie un mesaj la *cout*, face o pauză de 2 secunde, golește stream-ul de ieșire și apoi mai întârzie 3 secunde până la încheiere:

```
#include <iostream.h>
#include <time.h>

void main(void)
{
    time_t start_time, current_time;

    cout << "Hello C++ world!" << endl;
    time(&start_time);

    do {
        time(&current_time);
    } while ((current_time - start_time) < 2);

    cout.flush();

    time(&start_time);

    do {
        time(&current_time);
    } while ((current_time - start_time) < 3);

    cout << "Done!" << endl;
}
```

În Capitolul 11 veți aprofunda stream-urile I/O prin buffer când veți examina clasele *streambuf* și *filebuf*.

## FOLOSIREA FUNCȚIILOR MEMBRU PENTRU CONTROLUL INDICATORILOR DE STREAM I/O

Ați învățat ceva mai devreme, în acest capitol, cum se controlează ieșirea unui stream I/O folosind manipulatorii *setiosflags* și *resetiosflags*. Pe lângă acești manipulatori, programele pot avea un control mai precis asupra parametrilor acestor indicatori folosind funcțiile membru ale stream-urilor I/O ce vor fi discutate în continuare. De exemplu, următorul program, *IOFLAGS.CPP*, folosește funcția membru *flags* pentru afișarea valorilor parametrului indicatorului de stream:

```
#include <iostream.h>
#include <iomanip.h>
```

```
void main(void)
{
    cout << "cout starting flag settings: " <<
        hex << cout.flags() << endl;
    cout << setiosflags(ios::showbase);
    cout << "cout ending flag settings: " << cout.flags() << endl;
}
```

După compilarea și execuția acestui program, pe ecran va apărea un rezultat dependent de compilatorul folosit, având următoarea formă generală:

```
C:\> IOFLAGS <ENTER>
cout starting flag settings: 1
cout ending flag settings: 0xc1
```

Folosind funcția membru *flags*, un program poate salva valorile unui indicator la diferite momente, pentru a putea fi ulterior readuse la forma inițială. De exemplu, programul SAVEFLAG.CPP, folosește funcția membru *flags* pentru a citi valoarea curentă a unui indicator și a o restabili ulterior.

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
{
    long flag_settings;

    flag_settings = cout.flags(); // Read the flags

    cout << "Hexadecimal values: " << hex << 255 << " " <<
        10 << endl;

    cout.flags(flag_settings); // Restore the flags

    cout << "Decimal values: " << 255 << " " << 10 << endl;
}
```

La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> SAVEFLAG <ENTER>
Hexadecimal values: ff a
Decimal values: 255 10
```

Pe lângă folosirea funcției membru *flags* pentru configurarea indicatorilor stream-urilor I/O, programele mai pot folosi și funcțiile membru *setf* și *unsetf*. Pentru a crea și a șterge parametrii indicatorilor folosind aceste două funcții membru, trebuie cunoscuți biții ce corespund fiecărui indicator. Prin folosirea tipului de enumerare al indicatorilor prezentați anterior în acest capitol, se pot determina valorile corespunzătoare ale biților. De exemplu, următorul program, SETCLEAR.CPP, folosește funcțiile membru *setf* și *unsetf* pentru a crea și a șterge indicatorii stream-ului I/O:

```
#include <iostream.h>
#include <iomanip.h>
```

```
void main(void)
{
    cout.setf(0x80); // Show base
    cout << hex << 255 << " " << 10 << endl;

    cout.unsetf(0x80); // Clear the base display
    cout << hex << 255 << " " << 10 << endl;
}
```

Programul folosește funcția membru *setf* pentru a activa afișarea în bază hexazecimală. După ce afișează două valori, programul folosește funcția membru *unsetf* pentru a dezactiva afișarea în baza respectivă. La compilarea și execuția programului, pe ecran va apărea:

```
C:\> SETCLEAR <ENTER>
0xff 0xa
ff a
```

## REZUMAT

Fiecare program pe care îl creați, indiferent de destinația sa, va folosi operații de intrare sau ieșire. De aceea, înțelegerea stream-urilor I/O și a posibilităților lor sunt esențiale pentru succesul în programarea C++. Înainte de a continua, în Capitolul 2, asigurați-vă că ați învățat următoarele:

- ✓ Imaginea optimă a unui stream I/O este o serie de octeți.
- ✓ Compilatorul folosește un fișier antet pentru a defini stream-urile I/O – *ios*, *istream* și *ostream*, precum și funcțiile și variabilele membru. Tipăriți o copie a acestui fișier și studiați-o cu atenție.
- ✓ Operatorii de inserție (<<) și de extracție (>>) permit programelor să introducă și să extragă caractere în/din stream-ul de I/O.
- ✓ Manipulatorii sunt elemente pe care le puteți plasa într-un stream de intrare sau de ieșire pentru a controla formatarea I/O. Fișierul antet *IOMANIP.H* definește manipulatorii disponibili.
- ✓ Când un program folosește un manipulator pentru a configura formatarea unui stream, C++ stabilește valoarea biților din câmpul indicatorului de stream ce controlează configurarea. Pentru a înțelege mai bine acea configurație de biți, consultați fișierul antet folosit de compilator la definirea stream-urilor.
- ✓ Stream-urile de intrare/ieșire conțin diferite metode (funcții) ce permit controlul formatului ieșirii sau realizarea unei anumite operații I/O. Compilatorul definește prototipurile funcțiilor membru ale stream-ului într-un fișier antet.
- ✓ Stream-urile de ieșire *cout* și *clog* realizează ieșirea prin buffer, ceea ce înseamnă că ieșirea este scrisă numai atunci când buffer-ul se umple, programul se încheie, programul golește buffer-ul, sau, în cazul lui *cout*, când programul realizează o operație de intrare.

## CAPITOLUL 2

### ACOMODAREA CU CLASE ȘI OBIECTE

Este posibil să puteți scrie o mulțime de programe C++ fără a înțelege cu adevărat sau fără a folosi obiecte, deși obiectele și clasele sunt esențiale pentru utilizarea la maximum a limbajului. Dar nu vă speriați! Acest capitol începe cu primii pași și prezintă regulile de bază ce trebuie cunoscute. Capitolele următoare adaugă concepte pe baza informațiilor prezentate aici. În acest mod, puteți învăța pe rând părțile importante ale noțiunii de obiect. Ca și mai înainte, trebuie să aveți răbdarea de a experimenta programele prezentate aici. Chiar și prin schimbări simple, puteți învăța foarte mult. La încheierea terminării capitolului, veți înțelege următoarele:

- ♦ Ce este un obiect
- ♦ Cum un program complicat, de mari dimensiuni, este mai ușor de conceput și de redactat folosind o abordare orientată pe obiect în locul uneia convenționale, orientată pe funcție
- ♦ Cum un obiect poate economisi timp și efort
- ♦ Ce este moștenirea
- ♦ Avantajul folosirii obiectelor
- ♦ Cum se creează un obiect
- ♦ Avantajele și dezavantajele funcțiilor *inline*
- ♦ Cum se folosesc două variabile diferite cu același nume în același program
- ♦ Ce sunt funcțiile constructor și destructor de clase
- ♦ Cum se atribuie valoarea unui obiect altui obiect

#### *PRIMUL CONTACT CU PROGRAMAREA ORIENTATĂ PE OBIECTE*

În cel mai simplu sens, un obiect este un lucru. Un câine, o carte, chiar un computer, toate sunt obiecte. În trecut, programatorii își concepeau programele ca pe liste lungi de instrucțiuni pentru realizarea unei sarcini specifice. La crearea



de programe orientate pe obiecte, accentul se pune pe obiectele componente ale programului.

Să presupunem, de exemplu, că scrieți un program ce implementează simplu un procesor. Dacă vă gândiți la toate funcțiile realizate de un procesor de texte, veți fi repede copleșit de acestea. În schimb, dacă vă imaginați procesorul de cuvinte ca o colecție de obiecte distincte, programul va deveni mai puțin intimidant. În figura 2.1 sunt ilustrate obiectele principale ale sistemului de procesare de texte.

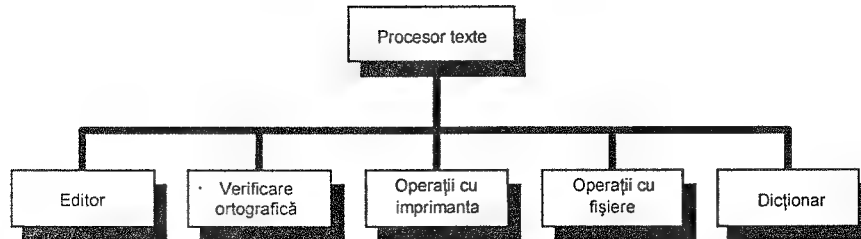


Figura 2.1. - Structura unui procesor de cuvinte ca o colecție de obiecte.

Pe măsură ce examinați fiecare obiect, veți descoperi că acesta este, de asemenea, compus din obiecte, cum se arată în Figura 2.2.

Când începeți să identificați obiectele din sistemul dvs., veți vedea că diferite părți ale programului folosesc același tip de obiecte. De aceea, scriind programul vostru în termeni de obiecte, veți putea ușor (și repede) să *refolosiți* codul scris pentru o anumită secvență într-o altă parte a programului, sau chiar în alt program. În aceasta rezidă o parte din puterea limbajului C++.

Primul pas în crearea programelor orientate pe obiecte este de a identifica obiectele ce formează sistemul. Pentru a înțelege mai bine cum se identifică obiectele sistemului, puteți face referință la următoarele cărți de analiză și proiectare orientate pe obiecte:

- *Object-Oriented Design and Applications* - Booch, Benjamin/Cummings, 1991
- *Object-Oriented Analysis*, Coad & Yourton, Yourdon Press, 1990
- *Object Data Management*, Cattel, Addison-Wesley, 1991
- *Object-Oriented Reuse, Concurrency, and Distribution*, Atkinson, Addison-Wesley, 1991
- *Object-Oriented Methods*, Graham, Addison-Wesley, 1991
- *An Introduction to Object-Oriented Programming*, Budd, Addison-Wesley, 1991

- *Object-Oriented Software Construction*, Meyer, Prentice-Hall, 1988
- *Object-Oriented System Analysis: A Model-Driven Approach*, Embley, Kurtz and Woodfield, Yourdon Press, 1992

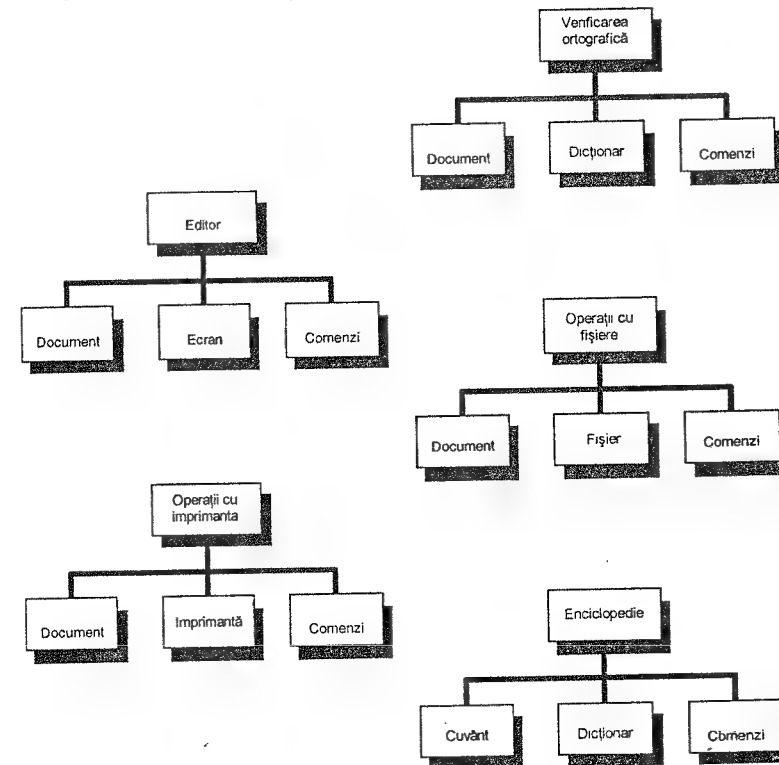


Figura 2.2. Identificarea obiectelor adiționale în cadrul procesorului de cuvinte.

După identificarea obiectelor, trebuie să determinați scopul fiecărui obiect. Pentru aceasta, gândiți-vă la operațiile realizate de un obiect sau la operațiile efectuate pe acel obiect. De exemplu, dacă *fișier* este un obiect, un program poate copia, șterge sau redenumi fișierul. Este important de observat că, în general, aceste operații se aplică fiecărui fișier de pe disc, indiferent de conținutul său. Aceste operații vor deveni *funcțiile membru* ale obiectului, pentru care ulterior veți scrie funcții C++ în programul respectiv. Apoi trebuie identificate informațiile despre obiect. În cazul unui obiect fișier, trebuie cunoscute numele fișierului, dimensiunea, gradul de protecție, eventual data la care fișierul a fost creat sau modificat ultima oară. Aceste date vor deveni *variabile membru* ale obiectului. Conceptual, obiectul fișier se poate vedea ca în Figura 2.3.



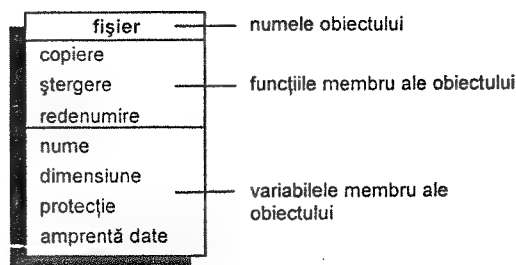


Figura 2.3. Funcțiile și variabilele membru ale unui obiect fișier.

**SĂ ÎNȚELEM PROGRAMAREA ORIENTATĂ PE OBIECTE**

Programarea orientată pe obiecte este bazată pe scrierea de programe cu referire la obiectele ce intră în componența unui sistem. În cadrul unui sistem, obiectele memorează tipuri specifice de informații și execută operații specifice pe acele informații. Primul pas în crearea unui program orientat pe obiecte este identificarea obiectelor sistemului, informațiile de bază referitoare la fiecare obiect și operațiile realizate pe obiecte. La scrierea primelor programe în C++, construcția lor se va opri, probabil, aici. Pe măsură ce deveniți mai experimentat, veți căuta relații între obiecte ce vor permite construcția unor obiecte noi din altele mai vechi. Pentru moment, însă, dacă cineva vă cere să definiți un obiect C++, spuneți-le pur și simplu că un obiect reprezintă o entitate a lumii reale, poate conține date (variabile membru) și are un set concret de operații (funcții membru) ce se efectuează asupra acestora.

Uneori este necesară afișarea sau tipărirea unor fișiere; alteori, fișierele conțin programe executabile de care aveți nevoie. Ca atare, la definiția obiectelor trebuie adăugate metodele de efectuare a operațiilor respective, ceea ce nu este greșit. Totuși, există o variantă mai bună de a prelucra cazurile speciale.

De exemplu, dacă fișierul conține un document, conținutul acestui fișier poate fi tipărit, operație care nu ar avea sens dacă fișierul ar conține un program executabil; în această ultimă situație execuția conținutului fișierului este operația dorită. La încercarea de a executa conținutul unui fișier document va rezulta o eroare. Soluția constă în identificarea a două noi clase obiect, denumite *document* și *program*. O modalitate de a crea aceste noi clase este adăugarea componentelor respective la fiecare clasă, așa cum se arată în Figura 2.4.

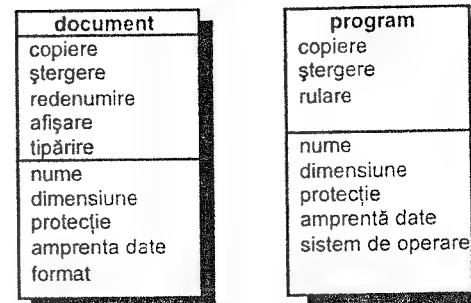


Figura 2.4. Crearea unor clase document și program independente.

După cum se poate vedea, la obiectul *document* se adaugă funcțiile membru *display* și *print* și variabila membru *format* (care specifică formatul intern al documentului, cum ar fi Word sau WordPerfect). În același mod, la obiectul *program* se adaugă funcția membru *run* și câmpul *sistem de operare*, care specifică dacă programul rulează sub DOS, WINDOWS sau UNIX.

Aceste două noi clase practic dublează clasa *fișier* definită anterior. De aceea o soluție mai bună este de a construi cele două clase bazate pe clasa *fișier*, cum se arată în Figura 2.5.

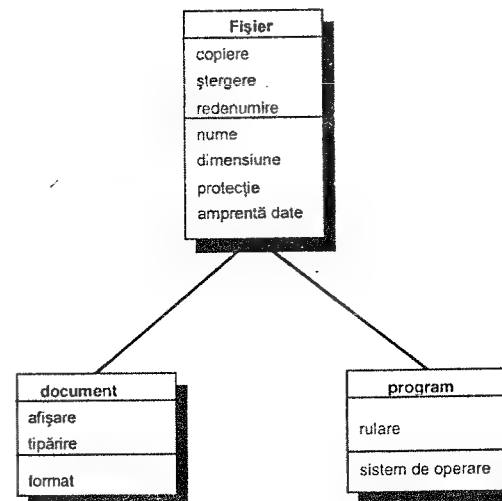



Figura 2.5. Crearea noilor clase pe baza clasei fișier.

Când programul dvs. construiește o clasă derivată dintr-o altă clasă, noua clasă moștenește funcțiile și variabilele membru ale clasei de bază. În cazul claselor *document* și *program*, obiectele create cu aceste clase pot să folosească nu


numai propriile funcții și variabile membru, ci și pe acelea ale clasei *fișier*. În acest mod, programul dvs. poate să execute cu ușurință afișarea, tipărirea, copierea, redenumirea sau ștergerea unui fișier document.



**SĂ SIMPLIFICĂM DEFINIȚIILE UNEI CLASE**

Pe măsură ce definiți clasele, încercați să dați definițiilor un caracter general. Cu alte cuvinte, variabilele și funcțiile membru care apar în clasa de bază trebuie să fie folosite de fiecare din clasele derivate (construite) din aceasta. Dacă este nevoie să adăugați la o clasă mai multe variabile și funcții membru, puteți defini o altă clasă, mai specializată, bazată pe clasa generală.

Construcția claselor în acest mod este esența *conceptului de moștenire a claselor*, concept ce va fi tratat în detaliu în Capitolul 4. S-ar putea să vă mirați de ce e nevoie de acest procedeu. Primul motiv pentru folosirea moștenirii este *refolosirea*, adică posibilitatea ca un program să folosească codul scris și testat pentru alt program. De exemplu, să presupunem că ați creat clasa *fișier* pentru un program de buget. Întrucât clasa există, aceasta poate fi rapid folosită în interiorul unor programe noi. Existența unei clase nu numai că diminuează efortul de programare, dar reduce și volumul testărilor necesare, întrucât funcțiile membru ale clasei *fișier* au fost testate și verificate anterior pentru programul de buget.



**SĂ ÎNȚELEM MOȘTENIREA**

Moștenirea este capacitatea unei clase *derivate* de a moșteni funcțiile și variabilele membru ale unei clase *de bază* existente. Pe măsură ce identificați și examinați obiectele ce compun un sistem, veți găsi deseori relații și asemănări între obiecte. În multe cazuri, aceste relații vă permit să construiți un obiect din altul, care conține toate metodele originalului. Când proiectați clasele obiect, încercați să dați claselor un caracter cât mai general. Dacă numărul funcțiilor și variabilelor incluse în definiția unei clase crește, atunci posibilitatea de a refolosi acea clasă, sau de a deriva alte clase din aceasta, scade.

## INGINERIA PROGRAMELOR ȘI FOLOSIREA OBIECTELOR

De ce ar trebui să ne bazăm pe obiecte în activitatea de programare? Există câțiva termeni de ingineria programelor care sunt folosiți frecvent pentru a descrie acest lucru. Cu toate că există dezacorduri în domeniul ingineriei de programe în privința folosirii optime a obiectelor, sunt clar recunoscute câteva avantaje oferite de acestea:

**Ușurința proiectării și refolosirii codului.** O dată ce codul funcționează corect, folosirea obiectelor mărește posibilitatea de a refolosi codul creat într-o aplicație pentru o altă aplicație.

**Fiabilitate crescută.** Deoarece bibliotecile de obiecte au fost anterior testate, folosirea codurilor existente (de lucru) va îmbunătăți fiabilitatea programului.

**Ușurința înțelegerii.** Permițând proiectanților și programatorilor să se concentreze asupra părților componente ale unui sistem și furnizând un cadru în care proiectanții pot identifica obiectele, operațiile efectuate cu acestea precum și informațiile pe care un obiect trebuie să le conțină, utilizarea obiectelor ajută programatorii să înțeleagă componentele importante ale unui sistem.

**Grad înalt de abstractizare.** Abstractizarea permite programatorilor să aibă o vedere de ansamblu, adică să ignore ansamblul temporar detaliile nesemnificative și să lucreze cu elementele de sistem ce sunt mai ușor de înțeles. De exemplu, prin concentrarea asupra obiectelor procesorului de cuvinte din exemplul anterior, implementarea acestuia a devenit mai puțin stresantă.

**Grad ridicat de încapsulare.** Încapsularea grupează toate părțile unui obiect într-un modul compact. De exemplu, clasa *fișier* definită în exemplele anterioare combină funcții și câmpuri de date cu care programul lucrează într-un fișier. Programatorul care lucrează cu clasa *fișier* nu trebuie să cunoască fiecare componentă a clasei, ci numai cum se folosește clasa în program. Clasa, la rândul ei, include toate componentele necesare.

**Ascunderea informației.** Ascunderea informației reprezintă capacitatea unui program de a trata o funcție, procedură, sau chiar un obiect, ca pe o „cutie neagră”, adică de a le folosi în anumite operații fără a cunoaște structura lor interioară. În Capitolul 1, de exemplu, programele au folosit obiecte tip stream I/O pentru intrare și ieșire, fără a necesita înțelegerea modului de funcționare a stream-urilor I/O.

Pe măsură ce vom examina conceptele de programare C++ în această carte, veți învăța care sunt legăturile dintre aceste concepte și definițiile de mai sus.

## SĂ ÎNȚELEM OBIECTELE ȘI CLASELE

În decursul discuției precedente, termenii *obiecte* și *clase* au fost folosiți foarte vag. În general, o *clasă* furnizează un șablon folosit de program la crearea obiectelor. De exemplu, clasa *fișier* descrisă anterior specifică variabilele și funcțiile membru ce vor fi folosite pentru obiectele *fișier* pe care programul le va crea ulterior. Un obiect, așadar, este o *instanță* (instance) a unui șablon de

clasă. Cu alte cuvinte, un obiect este o entitate, un exemplu specific, un articol cu care programul lucrează.

Definiția unei clase este similară unei definiții de structură în C. De exemplu, următoarele instrucțiuni definesc clasa *fișier*:

```
class file {
public:
    char filename[64];
    long size;
    unsigned protection;
    long datetime;
    int copy(char *target_name);
    int rename(char *target_name);
    int delete_file(void);
};
```

După cum se poate vedea, definiția clasei indică funcțiile și variabilele membru. Observați eticheta *public* ce apare la începutul clasei. Singurul mod în care programele pot avea acces direct la membrul unei clase, fie aceasta funcție sau variabilă, este ca acel membru să fie precedat de eticheta *public*. Mai târziu, în acest capitol, veți învăța cum să folosiți etichetele *public* și *private* pentru a controla modul în care programele pot avea acces la membrii unei clase. În Capitolul 4, veți învăța cum clasele condiționate folosesc eticheta *protected* pentru a furniza o cale intermediară de acces la membri. Deocamdată, însă, plasați eticheta *public* la începutul fiecărei clase; astfel toți membrii clasei vor deveni disponibili în tot programul.

O clasă definește un șablon pentru crearea ulterioară a obiectelor. Clasa însăși nu creează un obiect. Pentru a crea un obiect, se declară variabile de tip obiect, cum este arătat mai jos:

```
void main(void)
{
    file source, target;    // Declare two file objects

    // Other statements here
}
```

### CHEIA SUCCESULUI



### DIFERENȚA ÎNTRE CLASE ȘI OBIECTE

Dacă citiți articole și cărți despre C++ și programarea orientată pe obiecte, veți întâlni termenii *clasă* și *obiect*. O *clasă* furnizează un șablon ce definește funcțiile și variabilele membru necesare obiectelor având tipul clasei. Un *obiect*, pe de altă parte, este o *instanță*, un exemplu concret, de fapt o variabilă obiect. O clasă trebuie definită înainte de declararea obiectului.

Pentru a declara o variabilă obiect, specificați tipul clasei, urmat de numele variabilei obiect, ca mai jos:

```
nume_clasa    nume_obiect;
```

Procesul creării unui obiect este adesea denumit *instanțierea unui obiect* sau *crearea unei mostre de obiect*.

### ANALIZA UNUI EXEMPLU COMPLET

Cel mai bun mijloc de a înțelege clasele și obiectele C++ este de a crea un program simplu. Următorul program, MOVIES.CPP, creează o clasă denumită *movie* apoi creează 2 obiecte de tip *movie*, cu numele *fugitive* și *sleepless*. Programul definește clasa după cum urmează:

```
class movie {
public:
    char name[64];
    char first_star[64];
    char second_star[64];
    void show_movie(void);
    void initialize(char *name, char *first, char *second);
};
```

Diagramă de etichetare:

- Variable membru: *name*, *first\_star*, *second\_star*
- Funcții membru: *show\_movie*, *initialize*

După cum se poate vedea, clasa *movie* folosește trei variabile membru și două funcții membru. După definiția clasei, programul trebuie să definească funcțiile membru *show\_movie* și *initialize*, astfel:

```
void movie::show_movie(void)
{
    cout << "Movie name: " << name << endl;
    cout << "Starring: " << first_star << " and " << second_star << endl << endl;
}

void movie::initialize(char *movie_name, char *first, char *second)
{
    strcpy(name, movie_name);
    strcpy(first_star, first);
    strcpy(second_star, second);
}
```

Definiția funcțiilor clasei este asemănătoare definiției funcțiilor standard, existând totuși două diferențe. Mai întâi, numele funcțiilor sunt precedate de numele clasei și de simbolul „::”:

Diagramă de etichetare:

- Nume clasă: *movie*
- Nume funcție: *initialize*

```
void movie::initialize(char *movie_name, char *first, char *second)
```

În al doilea rând, instrucțiunile din corpul unei funcții ale unei clase pot face referință directă la variabilele membru ale clasei:

```
void movie::initialize(char *movie_name, char *first, char *second)
{
    strcpy(name, movie_name);
    strcpy(first_star, first);
    strcpy(second_star, second);
}
```

Nume variabilă locală funcție

Nume membri clasă

Următoarele instrucțiuni implementează programul MOVIES.CPP:

```
#include <iostream.h>
#include <string.h>

class movie {
public:
    char name[64];
    char first_star[64];
    char second_star[64];
    void show_movie(void);
    void initialize(char *name, char *first, char *second);
};

void movie::show_movie(void)
{
    cout << "Movie name: " << name << endl;
    cout << "Starring: " << first_star << " and " << second_star << endl;
}

void movie::initialize(char *movie_name, char *first, char *second)
{
    strcpy(name, movie_name);
    strcpy(first_star, first);
    strcpy(second_star, second);
}

void main(void)
{
    movie fugitive, sleepless;

    fugitive.initialize("The Fugitive", "Harrison Ford",
        "Tommy Lee Jones");
    sleepless.initialize("Sleepless in Seattle", "Tom Hanks",
        "Meg Ryan");

    fugitive.show_movie();
    sleepless.show_movie();
}
```

După cum se poate vedea, programul creează două obiecte de tipul *movie*:

```
movie fugitive, sleepless;
```

Nume clasă

Nume obiecte

Accesul la membrii unei clase este asemănător accesului la componentele unei structuri în C: se specifică numele obiectului, urmat de operatorul '.' și de numele membrului corespunzător. De exemplu, pentru a apela funcția *initialize*, obiectul *fugitive* folosește operatorul '.' și numele funcției membru, cum este arătat mai jos:

```
fugitive.initialize("The Fugitive", "Harrison Ford", "Tommy Lee Jones");
```

Nume obiect

Parametri funcție

Nume funcție

În acest caz, programul a folosit funcția membru *initialize* pentru a inițializa variabilele membru ale clasei. Mai târziu în acest capitol veți învăța cum să folosiți funcțiile *constructor* pentru a inițializa variabilele membru într-un mod mai natural.



### CÂND SE FOLOSESC CLASELE ȘI CÂND STRUCTURILE

Dacă ați programat în C, probabil ați recunoscut faptul că sintaxa pentru lucrul cu clase este similară celei folosite în C pentru structuri. V-ați putea întreba, când trebuie să folosiți clase, și nu structuri sau uniuni. După cum știți, acestea permit programului să memoreze date dependente. Programele dvs. ar trebui să folosească clase ori de câte ori asupra datelor au loc operații concrete. De exemplu, dacă aveți nevoie de o dată, atunci puteți folosi o structură. Dar dacă doriți ca programul să formateze și să afișeze data, să treacă data într-un fișier, sau să compare două date, este necesară utilizarea unei clase. Similar, dacă trebuie să optați între o structură și o uniune, decizia trebuie luată în funcție de numărul de valori pe care structura de date le conține la un moment dat. În sfârșit, țineți cont că, în mod prestabilit, membrii unei clase sunt de tip *private*, iar membrii unei uniuni sau structuri sunt de tip *public*.

Dacă experimentați structurile din C++, veți descoperi că ele suportă multe din conceptele clasei din C++, precum date publice și private, funcții membru etc. Ca o regulă, dacă creați obiecte, atunci folosiți clase.

### ACCESUL LA MEMBRII UNEI CLASE

În programul precedent, ați folosit operatorul de acces '.' pentru a apela funcțiile membru *initialize* și *show\_movie*. Când programul plasează membrii unui obiect după eticheta *public*, el poate avea acces la membrii acestuia

folosind operatorul '.', De exemplu, următorul program, PUBLIC.CPP, folosește funcția de inițializare pentru a atribui valori membrilor obiectelor *fugitive* și *sleepless*. Programul afișează apoi valorile diferiților membri, făcând referință la membri prin operatorul '.':

```
#include <iostream.h>
#include <string.h>

class movie {
public:
    char name[64];
    char first_star[64];
    char second_star[64];
    void show_movie(void);
    void initialize(char *name, char *first, char *second);
};

void movie::show_movie(void)
{
    cout << "Movie name: " << name << endl;
    cout << "Starring: " << first_star << " and " << second_star << endl << endl;
}

void movie::initialize(char *movie_name, char *first, char *second)
{
    strcpy(name, movie_name);
    strcpy(first_star, first);
    strcpy(second_star, second);
}

void main(void)
{
    movie fugitive, sleepless;

    fugitive.initialize("The Fugitive", "Harrison Ford",
        "Tommy Lee Jones");
    sleepless.initialize("Sleepless in Seattle", "Tom Hanks",
        "Meg Ryan");

    cout << "The last two movies I've watched are: " <<
        fugitive.name << " and " << sleepless.name << endl;

    cout << "I thought " << fugitive.first_star << " was great!" <<
        endl;
}
```

Deoarece membrii clasei sunt *public*, programul are acces direct la aceștia. Când compilați și executați acest program, ecranul va afișa următoarele:

```
C:\> PUBLIC <ENTER>
The last two movies I've watched are: The Fugitive and Sleepless in Seattle
I thought Harrison Ford was great!
```

Când o clasă definește variabilele membru ca fiind *public*, programul le poate referi folosind operatorul '.'. Totuși, așa cum veți afla mai târziu, acest acces direct la variabilele membru nu este întotdeauna de dorit.

### FOLOSIREA FUNCȚIILOR INLINE

Așa cum ați învățat, o clasă conține variabile și funcții membru. Când definiți funcțiile unei clase, aveți două opțiuni. Mai întâi, puteți defini funcțiile în afara definiției clasei, ca mai jos:

```
class movie {
public:
    char name[64];
    char first_star[64];
    char second_star[64];
    void show_movie(void);
    void initialize(char *name, char *first, char *second);
};

void movie::show_movie(void)
{
    cout << "Movie name: " << name << endl;
    cout << "Starring: " << first_star << " and " << second_star <<
        endl << endl;
}

void movie::initialize(char *movie_name, char *first, char *second)
{
    strcpy(name, movie_name);
    strcpy(first_star, first);
    strcpy(second_star, second);
}
```

În acest caz, definiția clasei trebuie să conțină prototipuri ce descriu fiecare funcție membru a clasei:

```
class movie {
public:
    char name[64];
    char first_star[64];
    char second_star[64];
    void show_movie(void);
    void initialize(char *name, char *first, char *second);
};
```

Prototipuri de funcții într-o clasă

## Succes cu C++

De asemenea, definițiile de funcții trebuie să specifice numele clasei înaintea numelui funcției:

```
      Nume clasă      Nume funcție
void movie::show_movie(void)
{
    cout << "Movie name: " << name << endl;
    cout << "Starring: " << first_star << " and " << second_star <<
        endl << endl;
}
```

În al doilea rând, se pot defini funcțiile membru ale unei clase în interiorul clasei, scriind instrucțiunile corpului funcției în cadrul declarației clasei. De exemplu, programul următor, `INLINE.CPP`, definește funcțiile membru ale clasei în interiorul declarației de clasă:

```
#include <iostream.h>
#include <string.h>

class movie {
public:
    char name[64];
    char first_star[64];
    char second_star[64];
    void show_movie(void)
    {
        cout << "Movie name: " << name << endl;

        cout << "Starring: " << first_star << " and " <<
            second_star << endl << endl;
    }

    void initialize(char *movie_name, char *first, char *second)
    {
        strcpy(name, movie_name);
        strcpy(first_star, first);
        strcpy(second_star, second);
    }
};

void main(void)
{
    movie fugitive, sleepless;

    fugitive.initialize("The Fugitive", "Harrison Ford",
        "Tommy Lee Jones");

    sleepless.initialize("Sleepless in Seattle", "Tom Hanks",
        "Meg Ryan");
}
```

## 2: Clase și obiecte

```
cout << "The last two movies I've watched are: " <<
    fugitive.name << " and " << sleepless.name << endl;

cout << "I thought " << fugitive.first_star << " was great!" <<
    endl;
}
```

După cum se observă, când o funcție membru este declarată inline, instrucțiunile funcției sunt scrise în interiorul clasei:

```
#include <iostream.h>
#include <string.h>


class movie {
public:
    char name[64];
    char first_star[64];
    char second_star[64];
    void show_movie(void)
    {
        cout << "Movie name: " << name << endl;
        cout << "Starring: " << first_star << " and " <<
            second_star << endl << endl;
    }

    void initialize(char *movie_name, char *first, char *second)
    {
        strcpy(name, movie_name);
        strcpy(first_star, first);
        strcpy(second_star, second);
    }
};
```

Declarații de funcții membru inline

Un avantaj al declarării funcțiilor membru inline este acela că întreaga clasă ocupă un loc compact în program. Din păcate, folosirea funcțiilor inline în acest mod mărește dimensiunea și complexitatea definițiilor clasei. Mai simplu spus, cu cât definiția clasei devine mai mare, cu atât devine mai dificil de înțeles. În plus, codul pentru funcțiile inline nu este partajat între tipurile de obiecte similare, așa cum se va vedea în continuare.

Când se definesc funcțiile membru în afara clasei, compilatorul C++ creează o copie a corpului funcției ce va fi ulterior folosită de fiecare obiect al acelei clase. Cu alte cuvinte, dacă se creează 1000 de obiecte, fiecare obiect va folosi o singură copie a codului funcției. O astfel de partajare a funcției este indicată, deoarece reduce semnificativ consumul de memorie al programului.



### DEFINIREA FUNCȚIILOR MEMBRU ÎN AFARA CLASEI

Când definiți funcțiile membru aveți două posibilități. Mai întâi, puteți defini funcțiile inline în interiorul clasei, astfel încât instrucțiunile funcției să apară în definiția clasei. În al doilea rând, puteți defini funcțiile în afara clasei. În majoritatea cazurilor, aceasta este varianta optimă, pentru a reduce dimensiunea și complexitatea clasei și de a asigura partajarea codului funcției între obiecte.

Dacă generați listing-ul în limbaj de asamblare al programelor PUBLIC.CPP și INLINE.CPP, veți vedea că compilatorul nu va partaja codul funcțiilor declarate inline.

### REZOLVAREA CONFLICTELOR DE NUME ÎNTRE MEMBRI ȘI PARAMETRI

Când transferați parametri unor funcții membru ale unei clase, numele parametrilor formali ai funcției creează variabile locale:

```
void movie::initialize(char *movie_name, char *first, char *second)
{
    strcpy(name, movie_name);
    strcpy(first_star, first);
    strcpy(second_star, second);
}
```

Nume parametri formali

După cum se observă, această funcție folosește nume diferite pentru parametri și pentru variabilele membru ale unei clase. Când numele unui parametru intră în conflict cu numele unui membru al unei clase, numele parametrului va fi folosit, iar numele membrului clasei va fi *ascuns*. Să presupunem, de exemplu, că funcția *initialize* ar folosi următoarele nume de parametri:

```
void movie::initialize(char *name, char *first_star,
    char *second_star)
{
    strcpy(name, name);
    strcpy(first_star, first_star);
    strcpy(second_star, second_star);
}
```

După cum se observă, ar fi imposibil pentru compilatorul C++ să determine care nume aparțin parametrilor și care aparțin membrilor clasei. De aceea, compilatorul va presupune că fiecare nume aparține parametrilor.

Deoarece nu putem găsi întotdeauna nume sugestive și variate pentru parametri și pentru membrii unei clase, se poate utiliza operatorul de rezoluție globală (::) din C++, pentru a elimina necesitatea numelor distincte. Când programul


vrea să se refere la un membru al unei clase, numele acestuia trebuie precedat cu numele clasei și simbolul '::', cum este arătat mai jos:

```
void movie::initialize(char *name, char *first_star,
    char *second_star)
{
    strcpy(movie::name, name);
    strcpy(movie::first_star, first_star);
    strcpy(movie::second_star, second_star);
}
```

Referință nume membru

Referință nume parametru

Scrierea numelui clasei și a simbolului '::' anterior numelui unui membru al acesteia permite decelarea referinței la membrul respectiv atât de către compilator, cât și de alți programatori care citească codul.



### REZOLVAREA CONFLICTELOR DE NUME

În cadrul funcțiilor membru ale unei clase pot apărea situații când numele membrilor clasei poate coincide cu numele parametrilor transmiși funcției. În mod prestabilit, C++ rezolvă astfel de conflicte de nume prin folosirea parametrului (variabila locală) și ascunzând existența variabilei membru a clasei. Pentru a preveni astfel de conflicte de nume, scrieți în fața numelui membrului clasei numele clasei urmat de simbolul '::', cum se arată mai jos:

```
void dogs::assign_dogs(char *breed, int height, int weight)
{
    strcpy(dogs::breed, breed);
    dogs::height = height;
    dogs::weight = weight;
}
```

În acest caz, numele precedate de *dogs::* corespund numelor membrilor clasei. Celelalte nume corespund variabilelor locale.

### SĂ ÎNȚELEM MEMBRII DE TIP PRIVATE AI UNEI CLASE

Fiecare din clasele examinate până acum aveau variabilele și funcțiile membre declarate de tipul *public*. În acest mod, programul poate avea acces la membrii clasei în orice punct al său (acolo unde clasa este în vigoare), folosind operatorul '.'. Din păcate, accesul la toți membrii unei clase în acest mod poate duce la erori și la alte probleme. Iată de ce.

După cum am discutat mai înainte, unul din avantajele folosirii obiectelor este că programatorii nu trebuie să înțeleagă obiectul pentru a-l folosi. Cu alte cuvinte, programatorii pot trata obiectele ca pe o "cutie neagră". Pentru a folosi



un obiect, un programator trebuie să știe numai scopul obiectului și câteva funcții membru. Așa după cum veți învăța, multe obiecte conțin variabile membru ce memorează informații importante.

De exemplu, să presupunem că scrieți un program pentru Pentagon ce controlează toate rachetele nucleare adăpostite în silozuri de pe teritoriul Statelor Unite. În acest caz, programul folosește obiectele *silo* într-un mod similar cu cel indicat mai jos:

```
class silo {
public:
    initialize(int missile_type, char *location);
    bombs_away(char *password);
    int missile_type;
    char location[64];
    int fire_missiles; // If 0 don't fire, if 1 fire
    char password[64] = "Hillary";
};
```

După cum se observă (din motive de securitate evidente), pentru a lansa o rachetă, programul trebuie să specifice o parolă. Dacă parola este corectă, indicatorul variabilei *fire\_missiles* primește valoarea 1, iar rachetele lansate. Dacă parola nu e corectă, variabila rămâne 0, și vom avea pace mondială. Din nefericire, deoarece toți membrii clasei au fost declarați de tip public, programul va avea acces liber la membrii săi. De exemplu, programul va putea folosi următoarea instrucțiune pentru a lansa rachetele, ignorând astfel funcția cu acces prin parolă *fire\_missiles*:

```
wyoming_silo.fire_missiles = 1;
```

Când folosiți obiecte, accesul la majoritatea variabilelor membru trebuie limitat la funcțiile membru. În acest mod, singura modalitate ca programul să aibă acces la variabila membru *fire\_missiles* este prin folosirea unei funcții membru, adică programul este forțat să joace după regulile impuse de creatorul său.

Pentru a restricționa accesul la membrii clasei, se pot folosi membri *private*, amplasând eticheta *private* în interiorul declarației de clasă. Următoarea clasă, de exemplu, restricționează accesul la mai mulți membri ai clasei *silo*, aplicându-le eticheta *private*:

```
class silo {
public:
    initialize(int missile_type, char *location);
    bombs_away(char *password);
private:
    int missile_type;
    char location[64];
    int fire_missiles; // 0-don't fire, 1-fire
    char password[64] = "Hillary";
};
```

Membri de tip public

Membri de tip private

În acest caz, programul va putea avea acces la funcțiile membru *initialize* și *bombs\_away* folosind operatorul '.', dar variabilele membru vor putea fi folosite numai prin intermediul acestor funcții. Dacă programul ar încerca să obțină acces direct la variabilele membru, compilatorul C++ va genera erori de sintaxă.

Următorul program, PRIVATE.CPP, modifică clasa *movie* descrisă anterior folosind membri de tip *private*:

```
#include <iostream.h>
#include <string.h>

class movie {
public:
    void show_movie(void);
    void initialize(char *name, char *first, char *second);
private:
    char name[64];
    char first_star[64];
    char second_star[64];
};

void movie::show_movie(void)
{
    cout << "Movie name: " << name << endl;
    cout << "Starring: " << first_star << " and " << second_star <<
        endl << endl;
}

void movie::initialize(char *movie_name, char *first, char *second)
{
    strcpy(name, movie_name);
    strcpy(first_star, first);
    strcpy(second_star, second);
}

void main(void)
{
    movie fugitive, sleepless;

    fugitive.initialize("The Fugitive", "Harrison Ford",
        "Tommy Lee Jones");
    sleepless.initialize("Sleepless in Seattle", "Tom Hanks",
        "Meg Ryan");

    fugitive.show_movie();
    sleepless.show_movie();
}
```

După cum se poate vedea, programul a aplicat eticheta *private* pentru toate cele trei variabile membru. La compilarea și execuția acestui program, pe ecran vor fi afișate următoarele:

```
C:\> PRIVATE <ENTER>
Movie name: The Fugitive
Starring: Harrison Ford and Tommy Lee Jones
```

```
Movie name: Sleepless in Seattle
Starring: Tom Hanks and Meg Ryan
```

Să observăm că acest program nu mai afișează mesajele programului precedent PUBLIC.CPP:


```
void main(void)
{
    movie fugitive, sleepless;

    fugitive.initialize("The Fugitive", "Harrison Ford",
        "Tommy Lee Jones");
    sleepless.initialize("Sleepless in Seattle", "Tom Hanks",
        "Meg Ryan");

    cout << "The last two movies I've watched are: " <<
        fugitive.name << " and " << sleepless.name << endl;

    cout << "I thought " << fugitive.first_star << " was great!" <<
        endl;
}
```

Deoarece variabilele membru nu mai sunt publice, programul nu mai poate avea acces direct la acestea prin operatorul `.`. Dacă se dorește accesul la aceste variabile, trebuie create funcții membru special în acest scop. În cazul când credeți că scrierea unor astfel de funcții membru e prea dificilă și preferați accesul direct, nu faceți altceva decât să măriți posibilitatea apariției unor viitoare erori.



### SĂ ÎNȚELEM MEMBRII DE TIP PRIVATE AI UNEI CLASE

Când declarați o clasă, puteți defini membrii ei ca *public* sau *private* (sau *protected*, cum veți afla în Capitolul 4). Membrii publici ai unei clase sunt disponibili în tot programul folosind numele obiectului și operatorul punct (*class.member*) sau operatorul de indirectare (*class\_ptr->member*). Membrii *private* ai unei clase, pe de altă parte, pot fi utilizați numai prin funcțiile membru ale clasei. Prin declararea într-o clasă a unor membri *private*, programele pot controla mai bine valorile atribuite membrilor clasei și modul în care aceste valori sunt folosite. În mod prestabilit, toți membrii unei


clase sunt de tip *private*. Membrii publici se pot specifica scriind în fața lor eticheta *public*. Dacă ulterior se dorește declararea unor membri *private*, trebuie inserată eticheta *private*, cum se arată în continuare:

```
class some_class {
public:
    void some_function(char *parameter);
    void some_other_function(int a, int b, int c);
private:
    int key_value;
    char password[64];
};
```

Membri tip *public*

Membri tip *private*

În acest caz, funcțiile membru ale clasei sunt publice, în timp ce variabilele membru ale clasei sunt nonpublice. Totuși, în funcție de clasă, pot exista funcții membre nonpublice, sau variabile membru publice. La proiectarea claselor trebuie determinat care dintre membri trebuie să fie de tip public, și care trebuie să rămână de tip *private*.



### SĂ ÎNȚELEM CAMUFLAJUL INFORMAȚIEI

Camuflajul informației este procesul de proiectare a funcțiilor sau a claselor drept „cutii negre”. Cu alte cuvinte, pentru a folosi o funcție sau o clasă, programatorul nu trebuie să cunoască toate amănuntele interne de funcționare ale cutiei, ci mai degrabă operația realizată de cutie și modelul de interfațare cu cutia. În programele C++, membrii nonpublici ai unei clase realizează camuflajul informației.

### SĂ ÎNȚELEM CONSTRUCTORII ȘI DESTRUCTORII DE CLASE

Când se creează obiecte, este câteodată nevoie de a alocă memorie pentru zonele tampon (buffers) folosite de obiecte. Să presupunem, de exemplu, că lucrați cu un obiect *fișier* care memorează numele fișierului într-un șir de caractere. Când creați obiectul pentru prima dată, veți dori ca obiectul să aloce memorie în mod dinamic pentru a memora șirul de caractere. Mai târziu, când obiectul va fi desființat, veți dori să eliberați memoria alocată. Pentru a ajuta programele să realizeze astfel de operații de fiecare dată când un obiect este construit și apoi distrus, limbajul C++ introduce funcțiile de tip *constructor* și *destructor*. O dată ce v-ați însușit acești termeni, veți vedea mai târziu că un constructor nu este altceva decât o funcție ce se execută automat de fiecare dată când creați un obiect, iar un destructor este o funcție ce se execută automat când obiectul este distrus.

Ați folosit mai devreme, în acest capitol, funcția membru *initialize* pentru a inițializa variabilele membru ale obiectelor aparținând clasei *movie*. Folosind o funcție constructor, puteți elimina necesitatea ca programele dvs. să apeleze funcții ca *initialize*. În schimb, la crearea obiectului programele vor specifica pur și simplu valorile parametrilor, ca mai jos:

```
movie fugitive("The Fugitive", "Harrison Ford",
    "Tommy Lee Jones");
movie.sleepless("Sleepless in Seattle", "Tom Hanks", "Meg Ryan");
```

La rândul său, C++ va apela automat funcția constructor. În cadrul constructorului, puteți atribui parametrii câmpurilor membru, la fel cum ați făcut când ați folosit *initialize*.

O funcție constructor este specială prin aceea că nu returnează o valoare și nu are tipul *void*. În al doilea rând, funcția constructor folosește același nume ca și clasa. În cazul clasei *movie*, funcția constructor va fi denumită tot *movie*.

```
class movie {
public:
    void show_movie(void);
    movie(char *name, char *first, char *second): ———— Funcție constructor
private:
    char name[64];
    char first_star[64];
    char second_star[64];
};
```

După cum se poate vedea, funcția nu specifică o valoare de întoarcere. Ca și la funcțiile membru ale unei clase, puteți defini funcția constructor fie în interiorul clasei (în linie), fie în afara acesteia. Următorul program, CONSTRUC.CPP folosește funcția constructor *movie* pentru inițializarea obiectelor:

```
#include <iostream.h>
#include <string.h>
```

```
class movie {
public:
    void show_movie(void);
    movie(char *name, char *first, char *second);
private:
    char name[64];
    char first_star[64];
    char second_star[64];
};
```

```
void movie::show_movie(void)
{
```

```
cout << "Movie name: " << name << endl;
cout << "Starring: " << first_star << " and " << second_star <<
    endl << endl;
```


Declarația funcției constructor

```
movie::movie(char *name, char *first_star, char *second_star)
{
    strcpy(movie::name, name);
    strcpy(movie::first_star, first_star);
    strcpy(movie::second_star, second_star);
}
```

```
void main(void)
{
    movie fugitive("The Fugitive", "Harrison Ford",
        "Tommy Lee Jones");
    movie sleepless("Sleepless in Seattle", "Tom Hanks", "Meg Ryan");

    fugitive.show_movie();
    sleepless.show_movie();
}
```

Folosind funcțiile constructor, programele dvs. pot inițializa obiectele la crearea acestora, eliminând astfel necesitatea unor funcții de inițializare separate.



### SĂ ÎNȚELEM FUNCȚIILE CONSTRUCTOR

O *funcție constructor* este o funcție membru a unei clase pe care C++ o execută automat de fiecare dată când creați un obiect al unei anumite clase. Funcția constructor este specială prin aceea că folosește același nume cu al clasei. De exemplu, pentru o clasă denumită *dogs*, funcția constructor se va numi tot *dogs*. În plus, funcția nu returnează nici un rezultat și nu este de tip *void*.

```
class dogs {
public:
    dogs(char *breed, int height, int weight); ———— Funcție constructor
    void show_dogs(void);
private:
    char breed[64];
    int height;
    int weight;
};
```

## SĂ ÎNȚELEM FUNCȚIILE DESTRUCTOR

Un destructor este o funcție ce se execută automat când obiectul este distrus. Ca și la funcția constructor, funcția destructor are același nume cu clasa. De asemenea, funcția destructor nu returnează vreo valoare și nu este de tipul *void*. Programele nu pot transfera parametri la o funcție destructor. Funcția destructor se deosebește de cea constructor prin faptul că este precedată de semnul tilda (-):

```
class dogs {
public:
    dogs(char *breed, int height, int weight);
    ~dogs(void);           Funcție destructor
    void show_dogs(void);
private:
    char breed[64];
    int height;
    int weight;
};
```

Următorul program, DESTRUCT.CPP, adaugă o funcție destructor la clasa *movie*. Când obiectul este distrus (în acest caz, când programul se încheie), este apelată funcția destructor:

```
#include <iostream.h>
#include <string.h>

class movie {
public:
    void show_movie(void);
    movie(char *name, char *first, char *second);
    ~movie(void);
private:
    char name[64];
    char first_star[64];
    char second_star[64];
};

void movie::show_movie(void)
{
    cout << "Movie name: " << name << endl;
    cout << "Starring: " << first_star << " and " << second_star << endl << endl;
}

movie::movie(char *name, char *first_star, char *second_star)
{
    strcpy(movie::name, name);
    strcpy(movie::first_star, first_star);
```

```
    strcpy(movie::second_star, second_star);
}

movie::~movie(void)
{
    cout << "In the movie destructor for " << name << endl;
}

void main(void)
{
    movie fugitive("The Fugitive", "Harrison Ford",
                  "Tommy Lee Jones");
    movie sleepless("Sleepless in Seattle", "Tom Hanks", "Meg Ryan");

    fugitive.show_movie();
    sleepless.show_movie();
}
```

La prima vedere, următoarea definiție de funcție pare încurcată:

```
movie::~movie(void)
{
    cout << "In the movie destructor for " << name << endl;
}
```

Pentru a înțelege antetul funcției, să-l examinăm pe porțiuni. Primul *movie*, începând din stânga, urmat de simbolul '::', specifică faptul că această funcție corespunde clasei *movie*. Apoi, caracterul tilda ('-') ne indică începutul definiției funcției destructor. Cunoscând acestea, veți putea deduce că numele *movie* va mai apărea odată deoarece destructorul folosește întotdeauna numele clasei. În sfârșit, programul dvs. nu poate transfera parametri către un destructor, de aceea lista lui de parametri este *void*.

După cum se vede, programul nu apelează în mod explicit destructorul. În schimb, o dată definit un destructor, acesta se execută automat când obiectul este distrus. De obicei, o funcție destructor va elibera memorie alocată anterior sau va salva informații pe un fișier de pe disc. În exemplul anterior, destructorul afișează un mesaj pe ecran, astfel încât la compilarea și execuția programului rezultatele afișate vor fi următoarele:

```
C:\> DESTRUCT <ENTER>
Movie name: The Fugitive
Starring: Harrison Ford and Tommy Lee Jones
```

```
Movie name: Sleepless in Seattle
Starring: Tom Hanks and Meg Ryan
```

```
In the movie destructor for Sleepless in Seattle
In the movie destructor for The Fugitive
```

După cum observați, ultimele două linii ale rezultatului sunt scrise de funcțiile destructor.



### SĂ ÎNȚELEM FUNCȚIILE DESTRUCTOR

Un destructor este o funcție ce se execută automat când obiectele unei anumite clase sunt distruse. Funcțiile destructor au același nume ca al clasei în care sunt definite, fiind precedate de semnul tilda (~). Un destructor nu returnează nici o valoare și nu este *void*. Programul nu poate transmite parametri unui destructor. Următoarea clasă, *employee*, folosește o funcție destructor:

```
class employee {
public:
    employee(char *name, int age);
    ~employee(void);
    void show_employee(void);
private:
    char name[64];
    int age;
};
```

Funcțiile destructor, ca și cele constructor, trebuie să fie membri de tip *public*.

### FOLOSIREA FUNCȚIILOR CONSTRUCTOR MULTIPLE

În Capitolul 5 veți examina suprapunerea funcțiilor, adică folosirea unor funcții diferite, dar cu același nume. În timpul compilării, compilatorul C++ determină ce funcție trebuie apelată, ținând cont de numărul de parametri sau de tipul valorii funcției. Când definiți funcții constructor într-un program, puteți specifica mai multe funcții, din care compilatorul va selecta funcția corectă în funcție de necesitățile programului. De exemplu, următorul program, MULTICON.CPP, folosește două funcții constructor pentru clasa *message*. Primul constructor atribuie mesajul specificat de parametru, în timp ce al doilea constructor folosește mesajul prestabilit „Hello, world”:

```
#include <iostream.h>
#include <string.h>

class message {
public:
    message(char *user_message);
    message(void);
    void show_message(void);
private:
    char secret_message[64];
```

```
};

message::message(char *user_message)
{
    strcpy(secret_message, user_message);
}

message::message(void)
{
    strcpy(secret_message, "Hello, world");
}

void message::show_message(void)
{
    cout << "The message is " << secret_message << endl;
}

void main(void)
{
    message greeting;
    message book("Success with C++");

    greeting.show_message();
    book.show_message();
}
```

Așa cum se observă, la crearea obiectului *greeting* nu se specifică parametri; în acest caz, programul apelează constructorul *message* care nu are parametri. Când se creează obiectul *book*, programul transmite un șir de caractere constructorului *message* care suportă un parametru de tip șir de caractere. La compilarea și execuția acestui program pe ecran vor apărea următoarele:

```
C:\> MULTICON <ENTER>
The message is Hello, world
The message is Success with C++
```

### FOLOSIREA ARGUMENTELOR PRESTABILITE LA FUNCȚIILE CONSTRUCTOR

În Capitolul 5 veți învăța cum să specificați parametrii prestabiliți pentru funcții. În cazul apelării unor funcții fără a specifica valori pentru fiecare parametru, se pot specifica valori prestabilite. Următorul program DEFPARAM.CPP, folosește un parametru prestabilit pentru funcția constructor a clasei *message*:

```
#include <iostream.h>
#include <string.h>

class message {
public:
    message(char *user_message = "Hello, world");
```

```

    void show_message(void);
private:
    char secret_message[64];
};

message::message(char *user_message)
{
    strcpy(secret_message, user_message);
}

void message::show_message(void)
{
    cout << "The message is " << secret_message << endl;
}

void main(void)
{
    message greeting;
    message book("Success with C++");

    greeting.show_message();
    book.show_message();
}

```

### UN ALT MOD DE A INIȚIALIZA MEMBRII UNEI CLASE

Așa cum ați învățat, funcțiile constructor vă ajută să inițializați membrii unei clase la crearea acesteia. La examinarea programelor C++ se poate întâlni o tehnică de inițializare mai specială. De exemplu, să presupunem că doriți să inițializați prin constructorul *counter* variabila *count* la 0, ca mai jos:

```

counter::counter(void)
{
    counter = 0;
    // Other statements
}

```

Așa cum se va dovedi, C++ permite inițializarea variabilelor membru ale unei clase prin intercalarea numelui variabilei și a valorii dorite între simbolul ":" și corpul de instrucțiuni al funcției:

```

counter::counter(void) : counter(0)
{
    // Other statements
}

```

Numele variabilei și valoarea inițială

Următorul program, CON\_INIT.CPP folosește acest tip de inițializare pentru atribuirea a trei variabile membru ale clasei *object* valorilor 1, 2 respectiv 3:

```

#include <iostream.h>

class object {
public:
    object::object(void);
    void show_object(void);
private:
    int a;
    int b;
    int c;
};

object::object(void) : a(1), b(2), c(3) { }; // Inicializarea membrilor clasei

void object::show_object(void)
{
    cout << "a contains: " << a << endl;
    cout << "b contains: " << b << endl;
    cout << "c contains: " << c << endl;
}

void main(void)
{
    object numbers;

    numbers.show_object();
}

```

La compilarea și execuția acestui program, pe ecran va apărea următorul rezultat:

```

C:\> CON_INIT <ENTER>
a contains 1
b contains 2
c contains 3

```

### ATRIBUIREA VALORII UNUI OBIECT ALTUI OBIECT

Așa cum într-un program putem atribui valorile unei variabile de tip *int* altei variabile de același tip, similar putem atribui valorile unui obiect către alt obiect. Veți vedea că C++ simplifică foarte mult atribuirea de valori obiectelor. De fapt, se folosește pur și simplu operatorul de atribuire (=). De exemplu, următoarea instrucțiune va atribui valoarea unui obiect *date* altui obiect:

```
work_day = today;
```

Presupunând că obiectul *date* conține membrii *month*, *day* și *year*, operatorul de atribuire va atribui automat valori tuturor membrilor obiectului. Ca urmare, instrucțiunea anterioară este echivalentă cu instrucțiunile următoare:

```
work_day.month = today.month;
work_day.day = today.day;
work_day.year = today.year;
```

Următorul program, ASSIGN.CPP, folosește operatorul de atribuire pentru atribuirii obiectului *movie* altui obiect:

```
#include <iostream.h>
#include <string.h>

class movie {
public:
    void show_movie(void);
    void initialize(char *name, char *first, char *second);
private:
    char name[64];
    char first_star[64];
    char second_star[64];
};

void movie::show_movie(void)
{
    cout << "Movie name: " << name << endl;
    cout << "Starring: " << first_star << " and " << second_star << endl << endl;
}

void movie::initialize(char *movie_name, char *first, char *second)
{
    strcpy(name, movie_name);
    strcpy(first_star, first);
    strcpy(second_star, second);
}

void main(void)
{
    movie fugitive, sleepless;
    movie date_choice;

    fugitive.initialize("The Fugitive", "Harrison Ford",
        "Tommy Lee Jones");
    sleepless.initialize("Sleepless in Seattle", "Tom Hanks",
        "Meg Ryan");

    fugitive.show_movie();
    sleepless.show_movie();

    date_choice = sleepless;
    cout << "The date choice is: ";
    date_choice.show_movie();
}
```

Așa cum se observă, programul atribuie valoarea obiectului *sleepless* către obiectul *date\_choice*. La compilarea și execuția programului, pe ecran va apărea:

```
C:\> ASSIGN <ENTER>
Movie name: The Fugitive
Starring: Harrison Ford and Tommy Lee Jones

Movie name: Sleepless in Seattle
Starring: Tom Hanks and Meg Ryan

The date choice is: Movie name: Sleepless in Seattle
Starring: Tom Hanks and Meg Ryan
```

În timpul lucrului cu obiecte, abilitatea de a putea atribui valorile unor obiecte altor obiecte va fi foarte convenabilă. În Capitolul 8 veți învăța cum să suprapuneți operatorii C++. Folosind suprapunerea operatorilor, se va putea testa dacă două obiecte sunt egale printr-o instrucțiune de forma:

```
if (date_choice == sleepless)
```

În acest caz, suprapunerea operatorului '=' va determina programul să compare valoarea fiecărui membru în parte pentru a determina dacă cele două obiecte sunt egale.

### OBIECTE ȘI FUNCȚII

În programe, obiectele și funcțiile se pot folosi la fel ca structurile și funcțiile. De exemplu, puteți transmite unei funcții un obiect prin valoare sau prin referință (în cazul când se dorește modificarea valorii unui membru). În mod asemănător, funcțiile pot returna obiecte. Următorul program, CLASSFUN.CPP transmite obiectul *message* către două funcții diferite. Prima funcție folosește apelul prin valoare pentru a afișa variabilele membru ale obiectului. A doua funcție utilizează apelul prin referință pentru a modifica variabilele membru:

```
#include <iostream.h>
#include <string.h>

class message {
public:
    message(char *user_message, char *owner);
    void show_message(void);
    char message_owner[64];
private:
    char secret_message[64];
};

message::message(char *user_message, char *owner)
{
```



```

    strcpy(secret_message, user_message);
    strcpy(message_owner, owner);
}

void message::show_message(void)
{
    cout << "The message owner is " << message_owner << endl;
    cout << "The message is " << secret_message << endl;
}

void some_function(message note)
{
    note.show_message();
}

void change_owner(message *note)
{
    strcpy(note->message_owner, "Fred");
}

void main(void)
{
    message book("Success with C++", "Kris");

    some_function(book);
    change_owner(&book);
    some_function(book);
}

```

După cum se observă, programul transmite obiectul *book* funcției *some\_function* care, la rândul ei, afișează membrul obiectului folosind funcția *show\_message*. În acest caz, funcția nu poate afișa valorile variabilelor membru deoarece *secret\_message* nu este public. Singurul mod prin care programul poate obține acces la o variabilă membru este de a folosi o funcție membru. Mai departe, transmițând prin referință obiectul *book* funcției *change\_owner*, funcția va putea modifica membrul *message\_owner*. Ca și înainte, funcția nu poate schimba membrul *secret\_message* deoarece acesta este *private*.

**CHEIA SUCCESULUI**



**FOLOSIREA OBIECTELOR ȘI A FUNCȚIILOR**

Obiectele din programe, ca și structurile, pot fi transmise funcțiilor drept parametri. De exemplu, dacă o funcție are nevoie să aibă acces la membrii unei clase, obiectul poate fi transferat funcției prin valoare. Dacă funcția trebuie să modifice un membru al clasei, obiectul trebuie transferat funcției prin referință. Să nu uităm că o funcție poate avea acces direct numai la membrii publici ai unei clase. Pentru a modifica un membru nonpublic al clasei, funcția trebuie să folosească o funcție membru a clasei.

În Capitolul 10 veți învăța cum se transferă clase funcțiilor, folosind referințe C++, eliminând astfel necesitatea de a aplica operatorul de indirectare (→) în cadrul unei funcții. În schimb, programele pot avea acces la membrii unei clase folosind operatorul punct (.).

**SĂ ÎNȚELEGEM MEMBRII UNEI CLASE**

În fiecare din programele anterioare, clasele au folosit tipuri simple de membri, ca *int*, *float* ș.a.m.d. Pe măsură ce clasele devin mai complexe, membrii lor pot fi tablouri, structuri, sau chiar alte clase sau pointeri la clase. De exemplu, următorul program, *NESTED.CPP*, include un pointer la clasa *date* în interiorul clasei *employee*:

```

#include <iostream.h>
#include <string.h>

class date {
public:
    date(int month, int day, int year);
    void show_date(void);
private:
    int month;
    int day;
    int year;
};

class employee {
public:
    employee(char *name, int age, int month, int day, int year);
    ~employee(void);
    void show_employee(void);
private:
    char name[64];
    int age;
    date *hire_date; // Nested class
};

date::date(int month, int day, int year)
{
    date::month = month;
    date::day = day;
    date::year = year;
}

void date::show_date(void)
{
    cout << month << '/' << day << '/' << year << endl;
}

```

```

employee::employee(char *name, int age, int month, int day,
int year)
{
    strcpy(employee::name, name);
    employee::age = age;
    hire_date = new date(month, day, year);
}

employee::~employee(void)
{
    delete hire_date;
}

void employee::show_employee(void)
{
    cout << "Name: " << name << endl;
    cout << "Age: " << age << endl;
    cout << "Hire date: ";
    hire_date->show_date();
}

void main(void)
{
    employee manager("Joe Smith", 33, 12, 25, 1994);

    manager.show_employee();
}

```

Dacă nu înțelegeți încă alocarea și dealocarea memoriei din constructorul și destructorul clasei *employee*, în Capitolul 7 se va examina alocarea memoriei în detaliu. Deocamdată, rețineți că membrii unei clase pot fi *tablouri* (*arrays*), *structuri* (*structures*) sau chiar *pointeri* la alte clase.

### FOLOSIREA UNUI TABLOU DE CLASE

Așa cum clasele pot conține un tablou de valori, programele pot avea tablouri ce conțin mai multe obiecte. De exemplu, următorul program, ARRAY.CPP, folosește un tablou ce conține 5 obiecte de tipul *date*:

```

#include <iostream.h>
#include <string.h>

class date {
public:
    date(int month, int day, int year);
    date(void);
    ~date(void);
    void show_date(void);
private:

```

```

    int month;
    int day;
    int year;
}

date::date(int month, int day, int year)
{
    date::month = month;

    date::day = day;
    date::year = year;
    cout << "In date constructor: ";
    show_date();
}

date::date(void)
{
    cout << "In date constructor with no date" << endl;
}

date::~date(void)
{
    cout << "In date destructor: ";
    show_date();
}

void date::show_date(void)
{
    cout << month << '/' << day << '/' << year << endl;
}

void main(void)
{
    date holidays[5];
    date christmas(12, 25, 94);
    date halloween(10, 31, 94);
    date fourth(7, 4, 94);
    date new_years(1, 1, 95);
    date birthday(9, 30, 94);

    holidays[0] = christmas;
    holidays[1] = halloween;
    holidays[2] = fourth;
    holidays[3] = new_years;
    holidays[4] = birthday;
}

```

După cum se observă, programul inițializează tabloul în *main*. Când se folosesc tablouri de obiecte, trebuie înțeles că C++ apelează funcțiile constructor și

destructor pentru fiecare element al tabloului. În acest caz, funcțiile constructor și destructor afișează un mesaj pentru a vă înștiința că au fost apelate. La compilarea și execuția programului, pe ecran se va afișa:

```
C:\> ARRAY <ENTER>
In date constructor with no date
In date constructor with no date
In date constructor with no date
In date constructor with no date
In date constructor with no date
In date constructor: 12/25/94
In date constructor: 10/31/94
In date constructor: 7/4/94
In date constructor: 1/1/95
In date constructor: 9/30/94
In date destructor: 9/30/94
In date destructor: 1/1/95
In date destructor: 7/4/94
In date destructor: 10/31/94
In date destructor: 12/25/94
In date destructor: 9/30/94
In date destructor: 1/1/95
In date destructor: 7/4/94
In date destructor: 10/31/94
In date destructor: 12/25/94
```

Constructori pentru  
elemente libere din tablou

Constructori de date  
individuale

Destructori pentru  
elemente libere din tablou

Destructori de date  
individuale

## REZUMAT

În acest capitol au fost prezentate mai multe aspecte ale claselor C++. Următoarele capitole vor examina în detaliu conceptele și resursele în care stă forța limbajului C++. Veți învăța cum să realizați operații de intrare/ieșire bazate pe obiecte, cum să folosiți moștenirea, cum să suprapuneți operatorii unor clase, cum clasele *template* simplifică declarația obiectelor și multe altele. Înainte de a continua, totuși, asigurați-vă că ați învățat următoarele:

- ✓ Un obiect reprezintă o entitate a lumii reale, poate conține date (variabile membru) și un set de operații (funcții membru) ce se realizează asupra acestora.
- ✓ Programarea convențională asimilează programele ca liste lungi de instrucțiuni ce îndeplinesc o anumită sarcină, în timp ce programele orientate pe obiecte se concentrează pe obiectele ce alcătuiesc programul. Dacă vă gândiți la toate funcțiile unui procesor de texte, de exemplu, veți dezarma rapid. În schimb, dacă vedeți procesorul de texte ca pe o colecție de obiecte distincte, programul va deveni manevrabil.

- ✓ Când scrieți un obiect (să zicem un obiect fișier) care are anumite caracteristici (tipărire, salvare, redenumire), puteți refolosi codul acelui obiect în altă secțiune, sau chiar în alt program, economisind astfel timp și efort.
- ✓ Principalele avantaje ale folosirii obiectelor sunt: ușurința în proiectare și refolosirea codului; fiabilitatea crescută, ușurința înțelegerii; abstractizarea sporită; încapsulare și camuflaj al datelor.
- ✓ Pentru a crea un obiect, se declară variabile de tipul acelui obiect. Pentru a declara o variabilă obiect, se specifică tipul clasei, urmat de numele variabilei obiect.
- ✓ Un avantaj al declarării funcțiilor membre *inline* este că întreaga clasă este localizată într-o porțiune precisă a codului programului. Din nefericire, acest lucru mărește dimensiunea și complexitatea definițiilor de clase, iar codul pentru funcții *inline* nu este partajat de obiectele similare.
- ✓ Pentru a rezolva conflictul între membrii clasei și parametri (variabilele locale) cu același nume, se folosește operatorul de rezoluție globală (::) la referirea membrului clasei.
- ✓ Un constructor este o funcție ce se execută automat de fiecare dată când se creează un obiect, eliminându-se nevoia de funcții de inițializare separate, iar un destructor este o funcție ce se execută automat când un obiect este distrus, eliberând de obicei memoria alocată. Ambele funcții au același nume ca al clasei, nu returnează valori, și nu sunt *void*. Destructorul este precedat de tilda (~).
- ✓ Pentru a atribui valoarea unui obiect către alt obiect, se folosește operatorul de atribuire (*work\_day.year = today.year*).

## CAPITOLUL 3

### ACOMODAREA CU FIȘIERELE

În Capitolul 1 ați examinat stream-urile I/O în C++. Așa cum ați văzut, compilatorul folosește un antet pentru definițiile claselor de stream-uri I/O, ale metodelor și ale variabilelor din aceste clase. Folosind manipulatorii și funcțiile membru, programele pot realiza o varietate de operații I/O. Acest capitol examinează operațiile cu fișiere I/O în detaliu. Veți învăța că, folosind stream-urile fișier, programele pot efectua intrări/ieșiri cu operatori de inserție sau extracție și cu funcțiile membru. La încheierea acestui capitol, veți înțelege toate operațiile cu fișiere, inclusiv următoarele:

- Ce trebuie specificat la deschiderea unui fișier
- Ce tipuri de clasă sunt definite în fișierul antet FSTREAM.H
- Cum se deschide un fișier
- Cum se realizează operațiile de intrare/ieșire cu fișiere, după deschiderea acestora
- Cum se închide un fișier
- Cum se realizează formatarea ieșirii
- Ce este un specificator al modului de deschidere și controlul modului în care programul deschide fișierul
- Cum se testează reușita fiecărei operații și modul în care programul poate detecta erorile și reacționa la acestea
- Ce sunt operațiile pe fișiere binare și cum se scriu sau se citesc informațiile despre structuri, tablouri sau valori reale
- Operații pe fișiere cu acces secvențial și acces aleator
- Cum se poziționează pointerii de fișier pentru operații de intrare sau ieșire aleatoare

#### SĂ ÎNȚELEGEM OPERAȚIILE I/O CU FIȘIERE

În Capitolul 1 s-au dat exemple de programe ce foloseau operații I/O cu stream-urile *cin* și *cout*. Pentru a se folosi aceste stream-uri, în programe se făceau pur și simplu referințe la numele acestora. Compilatorul C++ asociază automat ecranul, respectiv tastatura, la cele două stream-uri. Dacă programele citesc informații dintr-un fișier, sau scriu informații într-un fișier, ele vor trebui, totuși,

să deschidă fișierul respectiv și să specifice dacă vor să citească dintr-un fișier de intrare sau să scrie într-un fișier de ieșire.

Pentru a înțelege mai bine stream-urile fișier, ar trebui examinat fișierul antet FSTREAM.H. Veți putea găsi aici definițiile claselor *ifstream*, *ofstream* și *fstream*. Programele ce folosesc fișiere vor crea obiecte folosind aceste tipuri de clase, în funcție de natura operației I/O ce trebuie realizată. De exemplu, pentru a realiza operații de intrare pe fișiere, veți folosi clasa *ifstream*. Analog, pentru operații de ieșire, se folosește clasa *ofstream*. În sfârșit, în cazurile în care programul trebuie să citească sau să scrie în/din același fișier, veți folosi clasa *fstream*.

### DESCHIDEREA UNUI FIȘIER PENTRU IEȘIRE

Modul cel mai simplu de a deschide un fișier este de a crea un obiect bazat pe una din clasele stream-urilor fișier enumerate anterior. De exemplu, pentru a deschide un fișier pentru ieșire, trebuie mai întâi declarat un obiect de tipul *ofstream*, ca mai jos:

Stream fișier de ieșire  
 ofstream output\_file;  
 Nume obiect fișier

Apoi, se poate folosi funcția membru *open* pentru a deschide fișierul:

Funcția membru open  
 output\_file.open("FILENAME.EXT");  
 Nume fișier

În acest caz, programul va deschide un fișier de ieșire cu numele FILENAME.EXT. După deschiderea fișierului de ieșire, se poate folosi operatorul de inserție pentru a scrie în fișier. De exemplu, următoarea instrucțiune va înscrie un mesaj în fișier:

```
output_file << "Acest mesaj este scris in fisier" << endl;
```

Când programul nu mai lucrează cu fișierul, trebuie folosită funcția membru *close* pentru a închide fișierul:

```
output_file.close();
```

Următorul program, BOOKINFO.CPP, scrie informații despre această carte în fișierul BOOKINFO.DAT:

```
#include <fstream.h>

void main(void)
{
    ofstream output_file;
```

```
output_file.open("BOOKINFO.DAT");

output_file << "Title: Success with C++" << endl;
output_file << "Author: Kris Jamsa" << endl;
output_file << "Publisher: Jamsa Press" << endl;
output_file << "Price: $29.95" << endl;

output_file.close();
}
```



### DESCHIDEREA UNUI FIȘIER PENTRU IEȘIRE

Pentru a scrie date într-un fișier, programul trebuie să creeze un obiect de tipul *ofstream*, care este definit în fișierul antet FSTREAM.H. Apoi, programul trebuie să deschidă stream-ul folosind funcția membru *open*. Pentru a scrie date în fișier, se folosește operatorul de inserție. După terminarea operațiilor, fișierul trebuie închis folosind funcția membru *close*. Următorul program ilustrează pașii necesari pentru a scrie date într-un fișier:

```
#include <fstream.h> // Include fișierul antet FSTREAM.H

void main(void) // Declară un obiect de tip ofstream
{
    ofstream budget;

    budget.open("BUDGET.DAT"); // Deschide fișierul dorit

    budget << "Payroll Information" << endl;
    budget << "XYZ Corporation" << endl; // Execută scrierea

    budget.close(); // Închide fișierul
}
```

### FOLOSIREA FUNCȚIEI CONSTRUCTOR PENTRU DESCHIDEREA UNUI FIȘIER

Așa cum ați învățat, programele pot deschide fișiere folosind funcția membru *open*. Pe lângă această funcție, programele pot folosi și funcția constructor a clasei pentru a deschide un fișier, cum se indică mai jos:

```
ofstream output_file("FILENAME.EXT");
```

Folosind funcția constructor în acest fel, se elimină instrucțiunea *open* din program. Ca o regulă generală, reducerea numărului de instrucțiuni într-un program duce la îmbunătățirea clarității programului. De aceea, în majoritatea programelor se va folosi funcția constructor din *ofstream* pentru deschiderea fișierelor.

## FOLOSIREA MANIPULATORILOR ȘI A FUNCȚIILOR MEMBRU DE IEȘIRE

Când se realizează operații pe fișiere utilizând stream-uri fișier, se pot folosi manipulatorii de ieșire prezentați în Capitolul 1. De exemplu, următorul program, HEXOCTDE.CPP, creează un fișier cu numele HEXOCTDE.DAT ce conține reprezentările în zecimal, octal și hexazecimal ale valorilor 1 până la 255:

```
#include <fstream.h>
#include <iomanip.h>

void main(void)
{
    int i;

    ofstream output("HEXOCTDE.DAT");

    output << " Dec " << " Oct " << " Hex " << endl;
    for (i = 0; i <= 255; i++)
        output << dec << setw(5) << i << setw(5) << oct << i <<
            setw(5) << hex << i << endl;

    output.close();
}
```

Pentru a folosi acești manipulatori, trebuie inclus fișierul antet IOMANIP.H, cum s-a arătat la începutul programului anterior.

Pe lângă folosirea manipulatorilor de stream I/O prezentați în Capitolul 1, se pot folosi și funcțiile membru ce corespund stream-urilor I/O. De exemplu, următorul program, ALPHABET.CPP, folosește funcția membru *put* pentru a scrie literele alfabetului în fișierul ALPHABET.DAT:

```
#include <fstream.h>

void main(void)
{
    ofstream alphabet("ALPHABET.DAT");

    for (int letter = 'A'; letter <= 'Z'; letter++)
        alphabet.put((char)letter);
}
```

În acest caz, programul folosește funcția membru *put* pentru a scrie valori de tipul *int* sub formă ASCII.

Următorul program, ABC\_INS.CPP, realizează o procesare identică, scriind caractere în fișier cu operatorul de inserție.

```
#include <fstream.h>

void main(void)
{
    ofstream alphabet("ALPHABET.DAT");
```

```
for (char letter = 'A'; letter <= 'Z'; letter++)
    alphabet << letter;
}
```

Să observăm că programul folosește variabile de tipul *char*. Dacă ar fi fost folosite variabile de tipul *int*, programul ar fi scris în fișier numerele de la 65 până la 90.

## REALIZAREA OPERAȚIILOR DE SCRIERE FORMATATĂ A FIȘIERELOR

Dacă sunteți familiarizați cu C, cunoașteți că un fișier poate fi scris cu diverse formătări, folosind funcția *sprintf*. Când programați în C++, puteți folosi manipulatorii prezentați în Capitolul 1 pentru a formata ieșirea. De exemplu, să presupunem că doriți să creați o tabelă de forma:

Name	Age	SSAN	Salary
Robert Jones	51	111-22-3333	\$55000.00
Betty Smith	43	333-22-1111	\$60000.00
Reggie Allen	30	111-11-0000	\$9000.00

Următorul program, TABLE.CPP, folosește manipulatorii de formatare pentru crearea tabelii anterioare:

```
#include <fstream.h>
#include <iomanip.h>

void main(void)
{
    ofstream report("EMPLOYEE.RPT");

    struct {
        char name[64];
        int age;
        char ssan[64];
        float salary;
    } employees[] = {{"Robert Jones", 51, "111-22-3333", 55000.00},
                    {"Betty Smith", 43, "333-22-1111", 60000.00},
                    {"Reggie Allen", 30, "111-11-0000", 9000.00}};

    report << "Name\t\tAge\tSSAN\t\tSalary" << endl;

    for (int i = 0; i < 3; i++)
    {
        report << setiosflags(ios::left) << setw(16) <<
            employees[i].name;
        report << setw(8) << employees[i].age << setw(16) <<
            employees[i].ssan;
        report << setprecision(2) << setiosflags(ios::right |
            ios::showpoint | ios::fixed);
        report << setw(8) << employees[i].salary << endl;
    }

    report.close();
}
```



### SCRIEREA FORMATATĂ A FIȘIERELOR

Pe măsură ce programele devin mai complexe, apare necesitatea realizării unei ieșiri formatare. Folosind manipulatorii de stream I/O prezentați în Capitolul 1, ca *setw*, *setprecision*, *setiosflags*, programele pot realiza formatarea scrierii fișierelor.

### CONTROLUL MODULUI DE DESCHIDERE A FIȘIERELOR DE IEȘIRE

În mod prestabilit, când se deschide un fișier pentru operații de ieșire, dacă există deja pe disc un fișier cu același nume, acesta din urmă va fi suprascris. În funcție de scopul programului, s-ar putea dori ca un fișier să fie extins prin adăugarea de informații în final, sau ca o operație pe un fișier să nu se realizeze dacă un fișier cu același nume deja există pe disc. Pentru a putea controla modul în care sunt deschise fișierele de ieșire, se poate specifica, la deschiderea acestora, un anumit mod de deschidere. De exemplu, următoarea instrucțiune deschide fișierul BUDGET.DAT pentru operații de extindere:

```
ofstream output("BUDGET.DAT", ios::app);
```

Dacă fișierul BUDGET.DAT există, programul va deschide fișierul, scriind noi informații în finalul acestuia. Dacă fișierul nu există, programul va crea fișierul. Următorul program, LOGFILE.CPP, deschide un fișier denumit LOGFILE.DAT pentru operații de extindere. Programul plasează informații în fișier, inserând în prealabil data și ora curentă. De exemplu, dacă doriți să urmăriți proiectele la care lucrați în cursul zilei, puteți apela LOGFILE astfel:

```
C:\> LOGFILE Formulate budget proposal <ENTER>
C:\> LOGFILE Start research for new CD-ROM product <ENTER>
```

În interiorul fișierului LOGFILE.DAT, intrările vor arăta astfel:

```
C:\> TYPE LOGFILE.DAT <ENTER>
11/09/93 17:07:38 Formulate budget proposal
11/09/93 17:07:50 Start research for new CD-ROM project
```

Următorul program implementează LOGFILE.CPP:

```
#include <fstream.h>
#include <time.h>

void main(int argc, char **argv)
{
    ofstream output_file("LOGFILE.DAT", ios::app);

    char time[9], date[9];
```

```
if (argc > 1)
{
    output_file << _strdate(date) << " " << _strtime(time);

    while (++argv) {
        output_file << " " << *argv;
    } while (*argv);

    output_file << endl;

    output_file.close();
}
```

După cum se poate vedea, programul deschide fișierul folosind specificatorul de mod de extindere *ios::app*. Programul folosește funcțiile de bibliotecă *\_strdate* și *\_strtime* pentru a obține data și ora, după care ciclează prin argumentele liniei de comandă, inserând fiecare argument pe aceeași linie cu data și ora curentă. În final, programul folosește funcția membru *close* pentru a închide fișierul.

### CARACTERE SPECIALE ȘI FIȘIERE I/O

În Capitolul 1 ați folosit caractere speciale ca *\n* (newline), *\t* (tab) și *\a* (bell ASCII) cu stream-ul de ieșire *cout*. Când programele realizează operații de ieșire cu fișiere, se pot folosi toate caracterele speciale prezentate în Capitolul 1. De exemplu, următorul program, HEX\_TOO.CPP, schimbă programul anterior HEXOCTDE.CPP pentru a crea o tabelă în hexazecimal, octal și zecimal ce folosește caracterul de tabulare (*\t*) pentru formatare:

```
#include <fstream.h>
#include <iomanip.h>

void main(void)
{
    int i;

    ofstream output("HEXOCTDE.DAT");

    output << "\tDec\tOct\tHex" << endl;
    for (i = 0; i <= 255; i++)
        output << dec << '\t' << i << '\t' << oct << i << '\t' << hex <<
            i << endl;

    output.close();
}
```

### ALȚI SPECIFICATORI AI MODULUI DE DESCHIDERE A FIȘIERELOR

Tocmai ați învățat că specificatorul de mod *ios::app* permite unui program să deschidă un fișier în modul extindere. Așa cum se arată în Tabela 3.1, funcția membru *open* suportă și alți specificatori de deschidere.

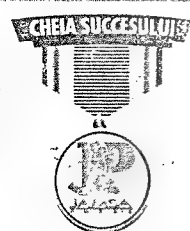


Modul de deschidere	Ce se realizează
ios::app	Deschide un fișier de ieșire pentru operații de extindere
ios::ate	Deschide un fișier pentru intrare sau ieșire, plasând pointerul de fișier la sfârșitul fișierului
ios::in	Deschide un fișier pentru intrare. Acest mod este prestabilit pentru obiecte de tip <i>ifstream</i>
ios::out	Deschide un fișier pentru ieșire. Acest mod este prestabilit pentru obiectele de tip <i>ofstream</i>
ios::nocreate	Deschide un fișier presupus existent. Dacă fișierul nu există, operația de deschidere nu se realizează
ios::noreplace	Creează un fișier nou. Dacă un fișier cu același nume există deja, operația de deschidere nu se realizează
ios::trunc	Șterge un fișier existent și creează un nou fișier
ios::binary	Deschide un fișier pentru operații de citire și scriere binară

**Tabela 3.1.** Modulurile de deschidere a fișierelor, așa cum sunt definite în fișierul *antet* folosit la definirea stream-urilor.

Dacă examinați conținutul fișierului *antet* folosit de compilator la definirea stream-urilor, veți găsi un tip enumerativ ce definește modulurile de deschidere a fișierelor, de forma următoare:

```
enum open mode {
    in = 0 x 01,    // Deschide un fișier pentru intrare
    out = 0 x 02,   // Deschide un fișier pentru ieșire
    ate = 0 x 04,   // Pozitionează pointerul de fișier
                    // la sfârșitul fișierului
    app = 0 x 08,   // Pozitionează pointerul unui fișier de ieșire
                    // la sfârșitul fișierului
    trunc = 0 x 10, // Trunchiază un fișier existent
    nocreate = 0 x 20, // Deschide un fis. existent sau nu deschide
    noreplace = 0 x 40, // Deschide un fis. nou sau op. nu se realizează
    binary = 0 x 80 // Deschide un fișier în mod binar
};
```



### CONTROLUL OPERAȚIILOR PE FIȘIERE

Când un program producează fișiere de ieșire, apar situații când informațiile trebuie scrise în continuarea unui fișier existent, sau când conținutul unui fișier nu trebuie șters. În astfel de situații, programele pot folosi specificatorii de mod în cadrul operației de deschidere. De exemplu, următoarea instrucțiune deschide un fișier pentru operații de extindere:

```
ofstream output("FILENAME.EXT", ios::app);
```

Analog, următoarea instrucțiune folosește indicatorul *ios::noreplace* pentru a preveni suprascrierea unui fișier existent:

```
ofstream output("FILENAME.EXT", ios::noreplace);
```

Folosind specificatorii modului de deschidere, programele pot prelua controlul asupra operațiilor de deschidere a fișierelor.

### DESCHIDEREA FIȘIERELOR PENTRU OPERAȚII DE INTRARE

Pentru a deschide un fișier pentru operații de intrare, programele creează, în mod normal, un obiect de tipul *ifstream*. Ca și în cazul când deschideți un fișier pentru ieșire, la deschiderea unui fișier pentru intrare se poate folosi atât funcția *open*, cât și funcția constructor a clasei *ifstream*. Prin urmare, fiecare din următoarele secvențe are ca efect deschiderea unui fișier denumit *BUDGET.DAT*:

```
ifstream input_file;          ifstream input_file("BUDGET.DAT");
input_file.open("BUDGET.DAT");
```

După ce un fișier este deschis pentru intrare, programul poate citi informații folosind operatorul de extracție sau funcțiile membru, cum se va arăta în continuare.

### REALIZAREA OPERAȚIILOR DE INTRARE PE FIȘIERE

După deschiderea unui fișier de intrare, conținutul său poate fi citit folosind funcțiile membru ale clasei *ifstream* sau operatorul de extracție. La citirea unui fișier, informațiile vor fi citite succesiv, de la începutul fișierului, până când va fi întâlnit marcajul de sfârșit de fișier. Pentru a determina dacă s-a ajuns la sfârșitul de fișier, programele pot folosi funcția membru *eof* într-un ciclu *while*, ca mai jos:

```
while (! input_file.eof())
{
    // Read and process the file
}
```

Următorul program, *SHOWFILE.CPP*, deschide pentru intrare un fișier specificat ca argument în linia de comandă, apoi citește și afișează conținutul său:

```
#include <fstream.h>

void main(int argc, char **argv)
{
    ifstream input(argv[1]);

    if (input.fail())
        cerr << "Error opening the file: " << argv[1] << endl;
    else
```

```
{
    while (!input.eof())
        cout.put((char)input.get());

    input.close();
}
```

**Observație:** Operatorul de mulare a tipurilor, (char), necesar în programul anterior pentru a evita un apel ambiguu al funcției put(), a fost înlăturat din exemplele ulterioare pentru a îmbunătăți claritatea acestora.

După cum se observă, programul folosește următorul ciclu pentru a afișa conținutul fișierului, literă cu literă:

```
while (!input.eof())
    cout.put(input.get());
```

Ciclul combină, de fapt, două instrucțiuni de citire și afișare a câte unui caracter:

```
letter = input.get();    // Read a character
cout.put(letter);        // Output the character
```

La întâlnirea sfârșitului de fișier, ciclul se termină, iar programul închide fișierul.

Pentru a afișa conținutul fișierului, programul SHOWFILE va fi apelat cu numele fișierului respectiv, ca mai jos:

```
C:\> SHOWFILE FILENAME.EXT <ENTER>
```

Dacă fișierul specificat nu poate fi deschis, programul detectează eroarea folosind funcția membru fail:

```
if (input.fail())
    cerr << "Error opening the file: " << argv[1] << endl;
```

În mod similar, programul MOREFILE.CPP citește și afișează conținutul fișierului, pagină cu pagină. După afișarea unei pagini de ecran, programul este pus în așteptare, până când utilizatorul apasă o tastă:

```
#include <fstream.h>
```

```
void main(int argc, char **argv)
{
```

```
    char line[128];
```

```
    long line_count = 0L;
```

```
    ifstream input(argv[1]);
```

```
    if (input.fail())
```

```
        cerr << "Error opening the file: " << argv[1] << endl;
    else
```

```
{
    while (!input.eof())
    {
        input.getline(line, sizeof(line));
        cout << line << endl;

        if ((++line_count % 24) == 0)
        {
            cout << "-MOREFILE-";
            cin.get();
        }
    }
    input.close();
}
```

După cum se poate vedea, programul deschide fișierul specificat pentru operații de intrare, folosind primul argument al liniei de comandă. Dacă are loc o eroare, programul afișează un mesaj de eroare și se încheie. În caz contrar, programul citează, citind conținutul fișierului prin funcția membru *getline*, până când este întâlnit sfârșitul de fișier. După cum se observă, programul folosește funcția membru *eof* pentru a detecta sfârșitul de fișier. După afișarea a 24 de linii de text, programul rămâne în așteptare până la apăsarea unei taste. Programul folosește operatorul % pentru a determina umplerea ecranului cu 24 de linii. De fiecare dată când numărul liniei curente împărțit la 24 dă restul 0, deja 24 de linii sunt afișate pe ecran, iar programul invită utilizatorul să apese o tastă. Pentru a afișa conținutul fișierului FILENAME.EXT pagină cu pagină, trebuie apelat programul MOREFILE astfel:

```
C:\> MOREFILE < FILENAME.EXT <ENTER>
```

#### CHEIA SUCCESULUI



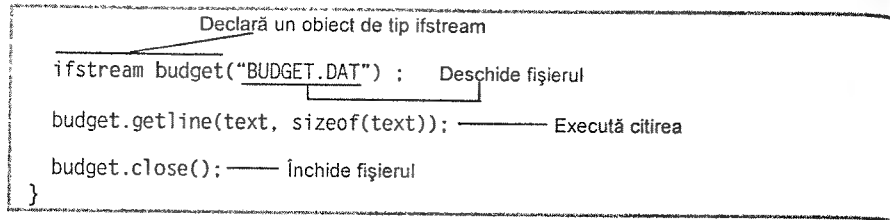
#### DESCHIDEREA UNUI FIȘIER PENTRU INTRARE

Pentru a citi informații dintr-un fișier, programul trebuie să creeze un obiect de tipul *ifstream*, definit în fișierul antet *FSTREAM.H*. Apoi, fișierul trebuie deschis prin funcția membru *open* sau prin funcția constructor. În continuare, programul poate folosi operatorul de extracție pentru a citi date din fișier. După terminarea prelucrării, fișierul trebuie închis folosind funcția membru *close*. Următorul program ilustrează pașii ce trebuie urmați pentru a citi date dintr-un fișier:

```
#include <fstream.h> // Include fișierul antet FSTREAM.H
```

```
void main(void)
```

```
{
    char text[256];
```



### TESTAREA REUȘITEI UNEI OPERAȚII I/O

Două din programele anterioare au folosit funcția membru *fail* pentru a determina dacă programul a putut deschide cu succes un fișier pentru intrare, cum se arată mai jos:

```

ifstream input(argv[1]);

if (input.fail())
    cerr << "Error opening the file: " << argv[1] << endl;
    
```

Așa cum ați învățat la Capitolul 1, funcția membru *fail* returnează valoarea 1 dacă a avut loc o eroare la ultima operație pe fișier. În acest caz, programul a detectat numai erorile la deschiderea fișierului. În Capitolul 1 ați învățat totuși că trebuie să testați reușita fiecărei operații de citire sau scriere. Următorul program, ASCICOPY.CPP, citește conținutul primului fișier specificat în linia de comandă, copiind conținutul acestuia în al doilea fișier al liniei de comandă.

```

#include <fstream.h>
#include <stdlib.h>

void main(int argc, char **argv)
{
    ifstream source(argv[1]);

    char line[128];

    if (source.fail())
        cerr << "Error opening the file: " << argv[1] << endl;
    else
    {
        ofstream target(argv[2]);

        if (target.fail())
            cerr << "Error opening the file: " << argv[2] << endl;
        else
        {
            while (! source.eof())
            {
                source.getline(line, sizeof(line));
                if (source.good())
                {
                    target << line << endl;
                    if (target.fail())
                        break;
                }
            }
        }
    }
}
    
```

```

{
    cerr << "Error writing the file: " <<
        argv[2] << endl;
    exit(1);
}

}
else if (! source.eof())
{
    cerr << "Error reading the file: " <<
        argv[1] << endl;
    exit(1);
}

}
source.close();
target.close();
}
}
    
```

După cum se observă, programul folosește funcțiile membru *good* și *fail* pentru a testa dacă fișierele au fost deschise cu succes și dacă operațiile de citire și scriere au reușit. La apariția unei erori, programul afișează un mesaj de eroare și folosește funcția *exit* din biblioteca sistemului pentru a se încheia. Așa cum se vede, programul testează mai întâi dacă operația de citire a reușit. În caz afirmativ, programul scrie datele în fișierul destinație.

După cum se observă, programul folosește funcția membru *good* pentru a testa reușita unei operații de citire. Așa cum se dovedește, când se întâlnește sfârșitul de fișier la citire, indicatorul *fail* este activ. Dacă programul nu testează atingerea sfârșitului de fișier în timpul detecției erorilor, atunci el ar putea trata finalul de fișier (care este un eveniment reușit) drept o eroare. Folosind funcția membru *good*, programul elimină această eroare posibilă.

**Observație:** Programul ASCICOPY.CPP va copia numai fișiere text ASCII, cum sunt fișierele programelor sursă C++. Dacă încercați să copiați un fișier binar, ca de exemplu un program executabil, operația de copiere nu se va realiza. Pentru a copia un fișier binar, trebuie să deschideți fișierul în mod binar, cum se va arăta în continuare.

Programele precedente au folosit funcția membru *fail* pentru a determina reușita sau eșecul unor operații I/O pe fișiere. Așa cum ați citit în Capitolul 1, programele pot folosi funcțiile membru din Tabela 3.2 pentru determinarea erorilor I/O pe fișiere:

Funcția membru	Destinația
good	Returnează 1 dacă operația precedentă a reușit
eof	Returnează 1 dacă se întâlnește sfârșitul de fișier
fail	Returnează 1 dacă a apărut o eroare
bad	Returnează 1 dacă s-a efectuat o operație incorectă

Tabela 3.2. Funcțiile membru ale stream-ului fișier pentru testarea erorilor.

Așa cum ați învățat în Capitolul 1, programele pot testa reușita unei operații I/O pe stream-uri folosind operatorul de exclamare (!). Astfel, următoarele instrucțiuni sunt identice:

```
if (input.fail())           if (!input)
```

În mod similar, următoarele două instrucțiuni testează ambele reușita unei operații I/O:

```
if (input.good())          if (input)
```

### REALIZAREA OPERAȚIILOR PE FIȘIERE BINARE

În mod prestabilit, operațiile pe fișiere sunt realizate în mod text. În funcție de destinația programelor, există situații când trebuie realizate operații pe fișiere binare. De exemplu, ați creat anterior programul ASCICOPY.CPP, care copia conținutul unui fișier în alt fișier. Așa cum s-a spus, acel program nu putea copia fișiere binare, cum sunt programele executabile. Și iată motivul:

Când un program citește un fișier în mod text, programul consideră valoarea ASCII 26 (CTRL-Z) drept sfârșit de fișier. Dacă programul încearcă să citească un fișier binar, este probabil ca programul să întâlnească această valoare undeva, în mijlocul fișierului, determinând ca funcția *eof* să ia valoarea *true*. Pentru a rezolva această problemă, fișierul trebuie deschis în mod binar, ca mai jos:

```
ifstream input(argv[1], ios::binary);
```

Pentru a realiza operații de intrare și ieșire, programele trebuie să folosească funcțiile membru *read* și *write*.

```
input.read(buffer, sizeof(buffer));
```

```
output.write(buffer, sizeof(buffer));
```

De exemplu, să presupunem că programul trebuie să scrie într-un fișier 30 de prețuri ale unor mărfuri. Următorul program, WRTSTOCK.CPP scrie valori reale în fișierul STOCKS.DAT folosind operații pe fișiere binare:

```
#include <fstream.h>
```

```
void main(void)
```

```
{
    int count;
```

```
    float price;
```

```
    ofstream stocks("STOCKS.DAT", ios::binary);
```

```
    if (stocks.fail())
        cerr << "Error opening the file STOCKS.DAT" << endl;
    else
    {
```

```
        for (count = 1; count <= 30; count++)
        {
            price = count * 100.0;
            stocks.write((char *) &price, sizeof(float));
        }
        stocks.close();
    }
}
```

Se observă că programul apelează funcția membru *write*, transferând funcției adresa valorii pentru a fi scrisă în fișier:

```
stocks.write((char *) &price, sizeof(float));
```

Dacă încercați să afișați conținutul fișierului STOCKS.DAT folosind comanda *TYPE*, operația va eșua, iar pe ecran vor apărea caractere neinteligibile. Amintiți-vă că fișierul STOCKS.DAT este fișier binar, nu fișier ASCII. Pentru a afișa prețurile mărfurilor se poate folosi programul RDSTOCK.CPP, prezentat în continuare:

```
#include <fstream.h>
#include <iomanip.h>
```

```
void main(void)
```

```
{
    float price;
```

```
    ifstream stocks("STOCKS.DAT", ios::binary);
```

```
    while (!stocks.eof())
```

```
    {
        stocks.read((char *) &price, sizeof(float));
```

```
        cout << setprecision(2) << setiosflags(ios::showpoint |
            ios::fixed) << price << endl;
```

```
    }
```

```
    stocks.close();
}
```

După cum se vede, programul ciclează, citind valorile reale din fișier, până la întâlnirea marcajului de sfârșit de fișier.

Ați creat mai devreme programul ASCICOPY.CPP, care copia conținutul unui fișier ASCII în alt fișier. Următorul program, BIN\_COPY.CPP, deschide fișierele sursă și destinație în mod binar, ceea ce permite programului să copieze atât fișiere text, cât și fișiere binare:

```
#include <fstream.h>
#include <stdlib.h>

void main(int argc, char **argv)
{
    ifstream source(argv[1], ios::binary);

    char line[1];

    if (source.fail())
        cerr << "Error opening the file: " << argv[1] << endl;
    else
    {
        ofstream target(argv[2], ios::binary);

        if (target.fail())
            cerr << "Error opening the file: " << argv[2] << endl;
        else
        {
            while (! source.eof() && ! source.fail())
            {
                source.read(line, sizeof(line));

                if (source.good())
                {
                    target.write(line, sizeof(line));

                    if (target.fail())
                    {
                        cerr << "Error writing the file: " <<
                            argv[2] << endl;
                        exit(1);
                    }
                }
            }
            else if (! source.eof())
            {
                cerr << "Error reading the file: " <<
                    argv[1] << endl;
                exit(1);
            }
        }
        source.close();
        target.close();
    }
}
```

După cum se observă, programul deschide ambele fișiere pentru operații binare folosind specificatorul de mod *ios::binary*. Programul folosește funcția membru *read* pentru a citi date de la intrare și funcția membru *write* pentru a scrie date

pe fișierul de ieșire. În plus, programul folosește funcțiile membru *fail* și *good* pentru a determina dacă operațiile I/O se termină cu succes. La întâlnirea marcajului la sfârșit de fișier, programul închide ambele fișiere.



### REALIZAREA OPERAȚIILOR PE FIȘIERE BINARE

Pe măsură ce informația memorată de programe devine mai complexă, apar situații când programele trebuie să citească și să scrie informații binare (ca de exemplu date de tip real sau structuri). Pentru a realiza operații pe un fișier binar, trebuie mai întâi deschis fișierul folosind specificatorul *ios::binary*. De exemplu, următoarea instrucțiune deschide un fișier de ieșire numit OUTPUT.DAT pentru operații binare:

```
ofstream data ("OUTPUT.DAT", ios::binary);
```

Pentru a realiza intrări/ieșiri în binar, programele vor folosi funcțiile membru *read* și *write*, specificând adresa de început a zonei de date precum și numărul de octeți ce urmează a fi introduși sau extrași:

```
input_file.read (& data, sizeof(data));
```

```
output_file.write (& data, sizeof(data));
```

După terminarea operațiilor de intrare/ieșire, programele trebuie să închidă fișierul folosind funcția membru *close*.

### ATENȚIE LA OPERAȚIILE DE INSERȚIE/EXTRACȚIE CU FIȘIERE BINARE

Așa cum ați văzut, operațiile de intrare/ieșire pe fișiere binare se efectuează prin funcțiile membru *read* și *write*. Dacă se folosesc operatorii de inserție/extracție pentru operații I/O pe fișiere binare, puteți întâlni erori neașteptate. Să considerăm, de exemplu, programul BAD\_OUT.CPP, care scrie câteva valori reale în fișierul BAD\_DATA.DAT folosind operatorul de inserție:

```
#include <fstream.h>

void main(void)
{
    ofstream output("BAD DATA.DAT", ios::binary);

    output << 100.0 / 11.1;
    output << 22.0 / 7.0;
    output << 100.0 / 11.1;

    output.close();
}
```

## Succes cu C++

Următorul program, BAD\_IN.CPP, citește, la rândul lui, conținutul fișierului BAD\_DATA.DAT, afișând valorile citite și valorile pe care fișierul ar trebui să le conțină:

```
#include <fstream.h>
```

```
void main(void)
```

```
{
    ifstream input("BAD_DATA.DAT", ios::binary);

    float value;

    input >> value;
    cout << value << " should be " << 100.0 / 11.1 << endl;
    input >> value;
    cout << value << " should be " << 22.0 / 7.0 << endl;
    input >> value;
    cout << value << " should be " << 100.0 / 11.1 << endl;

    input.close();
}
```

La compilarea și execuția acestui program, pe ecran vor apărea rezultate similare cu următoarele:

```
C:\> BAD_IN <ENTER>
9.009001 should be 9.009001
0.142869 should be 3.14286
0.00901 should be 9.009001
```

După cum se observă, valorile citite din fișier nu corespund valorilor scrise inițial în fișier. Și iată de ce. În mod prestabilit, când programele efectuează ieșiri pe flux, funcțiile stream-ului de ieșire convertesc valorile de diferite tipuri în reprezentări ASCII. De aceea, dacă am dori să vizualizăm conținutul fișierului BAD\_DATA.DAT folosind comanda TYPE, pe ecran ar apărea:

```
C:\> TYPE BAD_DATA <ENTER>
9.0090010.142869.009001
9.009001 3.14286 9.009001
```

Când un program va citi mai târziu valorile din acest fișier, el va încerca să convertească caracterele ASCII în valori reale. Din nefericire, așa cum s-a văzut la execuția programului, această conversie nu se realizează corect.

Dacă totuși doriți să folosiți operatorii de inserție/extracție cu fișierele binare, trebuie să suprapuneți acești operatori pentru a folosi funcțiile *read* și *write* descrise anterior. Din păcate, C++ va permite suprapunerea operatorilor de inserție/extracție numai pentru tipuri de date noi, definite în program. Următorul

program, BIN\_OPS.CPP, suprapune operatorul de inserție pentru valori de tipul *hidefloat*. În acest fel programul reușește să prelucreză fișierele binare folosind operatorii de inserție/extracție:

```
#include <fstream.h>

struct hidefloat { float data; };

ostream& operator <<(ostream& file, hidefloat value)
{
    file.write((char *) &value.data, sizeof(float));
    return(file);
}

void main(void)
{
    ofstream output("BAD_DATA.DAT", ios::binary);

    hidefloat value;

    value.data = 100.0 / 11.1;
    output << value;

    value.data = 22.0 / 7.0;
    output << value;

    value.data = 100.0 / 11.1;
    output << value;

    output.close();
}
```

De fiecare dată când compilatorul C++ întâlnește o valoare de tipul *hidefloat* asociată cu operatorul de inserție și cu un stream fișier de ieșire, programul va apela funcția operator ce folosește, la rândul ei, funcția membru *write* pentru a scrie valoarea binară în fișier.

Pentru citirea unui fișier binar, se poate folosi următorul program, READ\_BIN.CPP, arătat mai jos:

```
#include <fstream.h>

struct hidefloat { float data; };

ifstream& operator >>(ifstream& file, hidefloat *value)
{
    file.read((char *) value, sizeof(float));
    return(file);
}

void main(void)
```

```
{
    ifstream input("BAD_DATA.DAT", ios::binary);

    hidefloat value;

    input >> &value;
    cout << value.data << " should be " << 100.0 / 11.1 << endl;
    input >> &value;
    cout << value.data << " should be " << 22.0 / 7.0 << endl;
    input >> &value;
    cout << value.data << " should be " << 100.0 / 11.1 << endl;

    input.close();
}
```

La întâlnirea unui pointer către o variabilă de tipul *hidefloat* asociată cu operatorul de extracție și un stream fișier de intrare, programul va apela funcția operatorului. Deoarece funcția schimbă valoarea membrului unei structuri, ea primește adresa structurii. Așa cum vedeți, folosirea operatorilor de inserție/extracție este posibilă și la fișiere binare, dar devine destul de încurcată și dificil de înțeles.

### DESCHIDEREA FIȘIERELOR PENTRU OPERAȚII DE CITIRE ȘI SCRIERE

Fiecare din programele anterioare a deschis fișiere numai pentru intrare sau numai pentru ieșire, dar nu pentru ambele tipuri de operații. Când se creează fișiere pentru baze de date sau alte operații bazate pe fișiere, apar situații când același fișier trebuie deschis atât pentru operații de intrare, cât și pentru operații de ieșire. În astfel de situații, trebuie creat un obiect fișier de tipul *fstream*, ca mai jos:

```
fstream database("DATABASE.DAT", ios::in | ios::out);
```

La deschiderea unui fișier atât pentru operații de intrare, cât și de ieșire, programul lucrează cu doi pointeri de fișier - unul pentru intrare și altul pentru ieșire. În multe situații, fișierele deschise pentru intrare și pentru ieșire sunt folosite pentru acces aleator, cum se va arăta în continuare.

### FIȘIERE CU ACCES ALEATOR

Toate programele prezentate până acum în acest capitol au realizat operații *secvențiale* pe fișiere, prin parcurgerea fișierului de la începutul acestuia spre capătul său. Operațiile cu *acces aleator*, pe de altă parte, nu pornesc obligatoriu de la începutul fișierului. În schimb, ele pot trece de la o locație la alta a fișierului, în orice ordine. Dacă s-a deschis un fișier atât pentru intrare, cât și pentru ieșire, programul poate deplasa pointerul de citire la o locație a fișierului, iar pointerul de scriere la altă locație. Pentru a deplasa pointerul fișierului de la o locație la alta a fișierului, un program poate folosi funcțiile membru *seekg* și *seekp*. Funcția *seekg* poziționează pointerul de intrare al fișierului (denumirea *seekg* derivă din cuvintele *seek* și *get* adică poziționare în vederea unei operații

de intrare pe fișier). Asemănător, funcția *seekp* poziționează pointerul fișierului pentru efectuarea unei operații de ieșire (*seek* și *put*). Formatele funcțiilor *seekg* și *seekp* sunt următoarele:

```
seekp(offset, from_position);
```

```
seekg(offset, from_position);
```

Parametrul *offset* indică un deplasament în octeți în cadrul fișierului, și poate fi un număr întreg, pozitiv sau negativ. Parametrul *from\_position* specifică locația din fișier în raport cu care se realizează deplasamentul. Tabela 3.3 indică valorile enumerative ce pot fi specificate pentru parametrul *from\_position*.

Valori enumerative	Poziția fișierului
ios::beg	De la începutul fișierului
ios::cur	De la poziția curentă a pointerului în fișier
ios::end	De la sfârșitul fișierului

**Tabela 3.3.** Valorile enumerative care specifică pozițiile de bază într-un fișier cu acces aleator.

Următoarea instrucțiune, de exemplu, deplasează pointerul de ieșire (*put*) al fișierului la sfârșitul fișierului:

```
data_file.seekp(0, ios::end);
```

Pentru a înțelege mai bine operațiile pe fișiere cu acces aleator, să analizăm programul ABC.CPP, care creează fișierul LETTERS.DAT ce conține literele A până la H, urmate de literele O până la Z:

```
#include <fstream.h>

void main(void)
{
    ofstream output("LETTERS.DAT");

    for (char letter = 'A'; letter <= 'H'; letter++)
        output << letter;

    for (letter = 'O'; letter <= 'Z'; letter++)
        output << letter;

    output.close();
}
```

Folosiți comanda TYPE pentru a afișa conținutul fișierului, ca mai jos:

```
C:\> TYPE LETTERS.DAT <ENTER>
ABCDEFGHOPQRSTUVWXYZ
```

În continuare, programul RANDOM.CPP deschide fișierul LETTERS.DAT pentru operații de citire și scriere. Pentru început, programul poziționează pointerul de scriere al fișierului pe poziția octetului ce urmează literei H din fișier. Programul



scrie apoi în fișier literele de la I până la Z. În continuare, pointerul de citire al fișierului este poziționat la începutul acestuia, după care programul citește și afișează conținutul fișierului:

```
#include <fstream.h>

void main(void)
{
    fstream letters("LETTERS.DAT", ios::in | ios::out);
    letters.seekp(8, ios::beg);

    for (char letter = 'I'; letter <= 'Z'; letter++)
        letters << letter;

    letters.seekg(0, ios::beg);

    while (! letters.eof())
        cout.put((char)letters.get());

    letters.close();
}
```

La utilizarea fișierelor cu acces aleator, pot apărea situații când este nevoie de a determina poziția curentă a pointerilor de citire și scriere ai fișierului. Pentru a obține aceste valori, se folosesc funcțiile membru `tellg` și `tellp`, cum este arătat mai jos:

```
input_position = file_object.tellg();
output_position = file_object.tellp();
```

### RECAPITULAREA MODURILOR DE DESCHIDERE A UNUI FIȘIER

Așa cum s-a văzut, programele deschid fișierele folosind specificatorii modului de deschidere a fișierelor. De exemplu, un fișier poate fi deschis în modul extensie folosind specificatorul `ios::app`. Următorul program, `ASCIAPPD.CPP`, adaugă conținutul primului fișier specificat în linia de comandă la sfârșitul celui de al doilea fișier:

```
#include <fstream.h>
#include <stdlib.h>

void main(int argc, char **argv)
{
    ifstream source(argv[1]);

    char line[128];

    if (source.fail())
        cerr << "Error opening the file: " << argv[1] << endl;
    else
```

```
{
    ofstream target(argv[2], ios::app);

    if (target.fail())
        cerr << "Error opening the file: " << argv[2] << endl;
    else
    {
        while (! source.eof())
        {
            source.getline(line, sizeof(line));

            if (source.good())
            {
                target << line << endl;

                if (target.fail())
                {
                    cerr << "Error writing the file: " <<
                        argv[2] << endl;
                    exit(1);
                }
            }
            else if (! source.eof())
            {
                cerr << "Error reading the file: " <<
                    argv[1] << endl;
                exit(1);
            }
        }
        source.close();
        target.close();
    }
}
```

După cum se observă, programul deschide fișierul de ieșire pentru operații de extindere folosind modul `ios::app`. Programul realizează apoi operații de citire/scriere standard, folosind funcțiile membru `fail` și `good` pentru a verifica reușita fiecărei operații. Pentru a extinde fișierul `WEEKLY.NTS` cu conținutul fișierului `TODAY.NTS` se poate apela programul `ASCIAPPD` astfel:

```
C:\> ASCIAPPD TODAY.NTS WEEKLY.NTS <ENTER>
```

Tot în acest capitol, ați creat mai înainte fișierul `ASCICOPY.CPP` care copiază conținutul unui fișier ASCII în alt fișier. Dacă al doilea fișier există deja, programul va suprascrie conținutul fișierului existent. În alte situații nu este recomandabilă o astfel de suprascriere (deci o distrugere) a fișierelor. Pentru a preveni suprascrierea fișierelor, se poate folosi specificatorul de mod de deschidere `ios::noreplace`, cum se arată mai jos:

```
ofstream output_file("FILENAME.EXT", ios::noreplace);
```

Următorul program, OVERWRIT.CPP, folosește specificatorul `ios::noreplace` pentru a preveni distrugerea unui fișier printr-o operație de copiere.

```
#include <fstream.h>
#include <stdlib.h>

void main(int argc, char **argv)
{
    ifstream source(argv[1], ios::binary);

    char line[1];


    if (source.fail())
        cerr << "Error opening the file: " << argv[1] << endl;
    else
    {
        ofstream target(argv[2], ios::binary | ios::noreplace);

        if (target.fail())
            cerr << "Error opening the file: " << argv[2] <<
                " or file exists" << endl;
        else
        {
            while (! source.eof() && ! source.fail())
            {
                source.read(line, sizeof(line));

                if (source.good())
                {
                    target.write(line, sizeof(line));

                    if (target.fail())
                    {
                        cerr << "Error writing the file: " <<
                            argv[2] << endl;
                        exit(1);
                    }
                }
            }
            else if (! source.eof())
            {
                cerr << "Error reading the file: " <<
                    argv[1] << endl;
                exit(1);
            }
        }
        source.close();
        target.close();
    }
}
```

După cum se observă, dacă fișierul destinație există, programul va afișa un mesaj de eroare și se va încheia. Programul poate fi modificat de așa natură încât să întrebe utilizatorul dacă un fișier existent poate sau nu să fie suprascris.



### SĂ ÎNȚELEGEM FIȘIERELE CU ACCES ALEATOR

Majoritatea programelor realizează intrări/ieșiri secvențiale, prelucrând fișierele de la început spre sfârșit. Operațiile pe fișiere cu acces aleator, pe de altă parte, permit programelor să efectueze citiri sau scrieri de pe orice locație a fișierului. Înainte de a începe operațiile, programul poziționează pointerul de fișier pe locația dorită folosind funcțiile membru *seekp* și *seekg*. Funcția *seekg* poziționează pointerul de intrare (*get*) al fișierului, în timp ce funcția *seekp* poziționează pointerul de ieșire (*put*) al fișierului. Ambele funcții specifică locația de poziționat folosind un deplasament față de începutul, sfârșitul sau poziția curentă a fișierului. După poziționarea pointerului de fișier, operațiile I/O pe fișiere se realizează prin funcțiile membru *read* sau *write*.

### REALIZAREA IEȘIRII LA IMPRIMANTĂ

Există situații când ieșirea trebuie efectuată nu pe fișiere, ci la imprimantă. În general, imprimanta poate fi tratată ca un fișier de ieșire care va fi deschis cu numele de dispozitiv PRN.

```
ofstream printer("PRN");
```

De exemplu, programul PRTFILE.CPP tipărește la imprimantă conținutul fișierului specificat de primul argument al liniei de comandă:

```
#include <fstream.h>
#include <stdlib.h>

void main(int argc, char **argv)
{
    char text[256];

    ofstream printer("PRN");

    ifstream file(argv[1]);

    if (file.fail())
        cerr << "Error opening the file " << argv[1] << endl;
    else
    {
        while (! file.eof())
        {
            file.getline(text, sizeof(text));

            if (file.good())
```

```

    {
        printer << text << endl;

        if (printer.fail())
        {
            cerr << "Error writing to printer" << endl;
            exit(1);
        }
    }
    else if (! file.eof())
    {
        cerr << "Error reading from file " << argv[1] << endl;
        exit(1);
    }
}
file.close();
printer.close();
}
}

```

## REZUMAT

Pe măsură ce programele devin mai complexe, ele vor scrie în și vor citi informații din fișiere. După cum s-a văzut, C++ tratează fișierele foarte asemănător stream-urilor I/O prezentate în Capitolul 1. Înainte de a continua cu Capitolul 4, asigurați-vă că ați învățat următoarele:

- ✓ Când deschideți un fișier, de fapt specificați dacă realizați operații de intrare sau ieșire.
- ✓ Fișierul antet `FSTREAM.H` definește tipurile de clase *ifstream*, *ofstream* și *fstream*. Clasa *ifstream* corespunde fișierelor de intrare. Clasa *ofstream* este folosită pentru fișiere de ieșire. În sfârșit, clasa *fstream* este folosită pentru fișiere pe care le deschideți pentru ambele operații.
- ✓ Pentru a deschide un fișier, se poate folosi funcția membru *open* sau funcția constructor a clasei.
- ✓ După deschiderea unui fișier, pentru a realiza operații de intrare/ieșire se pot folosi atât operatorii de inserție și extracție, cât și funcțiile membru.
- ✓ După ce un fișier a fost folosit, el trebuie închis cu funcția membru *close*.
- ✓ Prin folosirea manipulatorilor de stream-uri de fișier ca *setw*, *setprecision*, *setiosflags*, programele pot realiza ieșirea

formatată pe fișiere. De asemenea, se pot scrie pe fișiere caractere speciale ca `\a`, `\t`, sau chiar `\n`.

La deschiderea unui fișier pentru intrare sau ieșire, programele pot folosi specificatorii modului de deschidere pentru a controla modul în care se deschid fișierele. De exemplu, folosind specificatorul *ios::app*, se poate deschide un fișier existent pentru a fi extins.

- ✓ La efectuarea operațiilor I/O pe fișiere sau la extinderea fișierelor, programele pot folosi funcții membru, ca *fail* sau *good*, pentru a testa succesul fiecărei operații. În cazul unei nereușite, programele o pot detecta și pot reacționa la eroare.
- ✓ Dacă programele scriu sau citesc din fișiere date de tip structuri, tablouri sau chiar date de tip real, ele trebuie să realizeze operații pe fișiere binare. Fișierele binare se deschid folosind modul *ios::binary*, iar operațiile pe acestea se efectuează cu funcțiile membru *read* și *write*.
- ✓ Operațiile secvențiale pe fișiere se desfășoară de la începutul spre sfârșitul fișierului. Pe de altă parte, operațiile cu acces aleator pe fișiere se pot efectua pe orice locație a unui fișier. Poziționarea pointerilor de fișier pentru operații de citire aleatoare se face cu funcția membru *seekg*. Analog, poziționarea pointerilor de fișier pentru operații de scriere aleatoare se realizează cu funcția *seekp*.

## CAPITOLUL 4

### ACOMODAREA CU MOȘTENIREA CLASELOR

În Capitolul 2 ați învățat că obiectele, care modelează realitatea înconjurătoare, pot avea caracteristici comune. Când definiți clasele într-un program, veți vedea că și ele pot avea aceleași proprietăți. Folosind acest avantaj al relațiilor dintre obiecte, se poate reduce substanțial cantitatea de cod care trebuie scrisă. Reducerea codului înseamnă nu numai reducerea timpului de programare, dar și îmbunătățirea lizibilității programului; o cantitate mai mică de cod conduce la erori mai puține.

Când construiți clase bazate pe relații între obiecte, folosiți conceptul de *moștenire* - o clasă derivată poate *moșteni* proprietățile clasei de bază. În mod normal, proiectarea începe cu o clasă simplă, din care se vor construi clase mai complexe. De exemplu, s-ar putea începe cu o clasă *engine* din care se poate construi o clasă *boat engine*, o clasă *car engine* sau *motorcycle engine*. Acest capitol examinează moștenirea claselor C++ în detaliu, iar în momentul încheierii sale veți cunoaște:

- ♦ Ce este și la ce folosește moștenirea
- ♦ Cum se folosesc funcțiile constructor în cazul moștenirii
- ♦ Care este „știința” și „arta” programării pe obiecte
- ♦ Ce este moștenirea multiplă
- ♦ Ce este moștenirea pe mai multe niveluri
- ♦ Ce este membrul protejat al unei clase și la ce folosește

#### CONCEPTUL DE MOȘTENIRE

Când definiți clase într-un program, apar situații când două sau mai multe clase au caracteristici comune (aceiași membri). În loc de a reproduce membrii în fiecare clasă (consumând timp și introducând o cantitate mare de cod care ar putea avea o mulțime de erori și care necesită foarte mult timp pentru a-l depăna), s-ar putea defini o clasă de bază care să conțină funcțiile membru comune. Apoi, s-ar putea deriva (construi) clase succesive din clasa de bază. Într-o astfel de abordare, clasele derivate vor moșteni caracteristicile clasei de bază.

# ANALIZA UNUI EXEMPLU SIMPLU

Să presupunem, de exemplu, că scrieți un program pentru un medic veterinar care dorește să țină înregistrările unor rase diferite de câini pe care îi tratează, precum și bolile de care suferă aceștia. În mod concret, veterinarul dorește să vadă ce tipuri de boli afectează câinii cu pete și pe cei fără pete. Un mod de abordare poate fi crearea a două clase de câini, o clasă pentru câinii cu pete și alta pentru cei fără pete, cum se arată mai jos:

```
class spotted_dogs {
public:
    spotted_dogs(char *breed, int height, int weight, char *color,
        char *spot_color);
    void show_breed(void);
    void spot_info(void);
private:
    char breed[64];
    int height;
    int weight;
    char color[64];
    char spot_color[64];
};

class unspotted_dogs {
public:
    unspotted_dogs(char *breed, int height, int weight,
        char *color);
    void show_breed(void);
private:
    char breed[64];
    int height;
    int weight;
    char color[64];
};
```

Următorul program, TWODOGS.CPP, folosește aceste două clase pentru a crea obiectele de tip *spotted\_dogs* și *unspotted\_dogs*:

```
#include <iostream.h>
#include <string.h>

class spotted_dogs {
public:
    spotted_dogs(char *breed, int height, int weight, char *color,
        char *spot_color);
    void show_breed(void);
    void spot_info(void);
private:
    char breed[64];
```

```
int height;
int weight;
char color[64];
char spot_color[64];
};

class unspotted_dogs {
public:
    unspotted_dogs(char *breed, int height, int weight,
        char *color);
    void show_breed(void);
private:
    char breed[64];
    int height;
    int weight;
    char color[64];
};

spotted_dogs::spotted_dogs(char *breed, int height, int weight,
    char *color, char *spot_color)
{
    strcpy(spotted_dogs::breed, breed);
    spotted_dogs::height = height;
    spotted_dogs::weight = weight;
    strcpy(spotted_dogs::color, color);
    strcpy(spotted_dogs::spot_color, spot_color);
}

void spotted_dogs::show_breed(void)
{
    cout << "Breed: " << breed << endl;
    cout << "Height: " << height << " Weight: " << weight << endl;
    cout << "Color: " << color << endl;
    cout << "Spot color: " << spot_color << endl << endl;
}

void spotted_dogs::spot_info(void)
{
    cout << breed << " has " << spot_color << " spots" << endl <<
        endl;
}

unspotted_dogs::unspotted_dogs(char *breed, int height, int weight,
    char *color)
{
    strcpy(unspotted_dogs::breed, breed);
    unspotted_dogs::height = height;
    unspotted_dogs::weight = weight;
    strcpy(unspotted_dogs::color, color);
```

```

    }

void unspotted_dogs::show_breed(void)
{
    cout << "Breed: " << breed << endl;
    cout << "Height: " << height << " Weight: " << weight << endl;
    cout << "Color: " << color << endl << endl;
}

void main(void)
{
    spotted_dogs happy("Dalmatian", 24, 60, "white",
        "black or brown (liver)");
    unspotted_dogs rover("Labrador Retriever", 24, 65,
        "black or yellow");

    happy.show_breed();
    happy.spot_info();

    rover.show_breed();
}

```

După cum se poate vedea, programul atribuie valori variabilelor membru, apoi afișează aceste valori folosind funcția *show\_breed*. La compilarea și execuția acestui program, pe ecran va apărea:

```

C:\> TWO DOGS <ENTER>
Breed: Dalmatian
Height: 24 Weight: 60
Color: white
Spot color: black or brown (liver)

Dalmatian has black or brown (liver) spots

Breed: Labrador Retriever
Height: 24 Weight: 65
Color: black or yellow

```

Dacă priviți cu atenție cele două clase, veți observa că ele conțin câțiva membri comuni. Folosind conceptul de moștenire, se poate crea o clasă ce conține membrii comuni, apoi alte două clase mai mici ce conțin membrii specifici pentru câinii cu pete și fără pete. Să creăm, pentru început, clasa *dogs* ce conține membrii comuni:

```

class dogs {
public:
    dogs(char *breed, int height, int weight, char *color);
    void show_breed(void);
private:

```

```

    char breed[64];
    int height;
    int weight;
    char color[64];
};

```

Să creăm acum două clase mai mici bazate pe clasa *dogs*. Cu alte cuvinte, două clase care *moștenesc* caracteristicile (membrii) clasei *dogs*. Următoarele instrucțiuni, de exemplu, creează clasele folosind moștenirea:

```

class spotted_dogs : public dogs {
public:
    spotted_dogs(char *breed, int height, int weight, char *color,
        char *spot_color);
    void show_breed(void);
    void spot_info(void);
private:
    char spot_color[64];
};

class unspotted_dogs : public dogs {
public:
    unspotted_dogs(char *breed, int height, int weight,
        char *color);
};

```

Observați semnul ":" situat după numele clasei și urmat de cuvintele *public dogs*. Semnul ":" informează compilatorul C++ că clasa definită moștenește caracteristicile clasei care urmează:

```

          Clasă derivată
          /
class spotted_dogs : public dogs {
                      \
                      Clasă de bază

```

După cum se observă, clasa *spotted\_dogs* adaugă noi funcții membru și o variabilă membru. Clasa *unspotted\_dogs* nu adaugă nici un membru nou, deoarece câinii fără pete nu au nevoie de alte caracteristici, în afara celor furnizate de clasa de bază *dogs*. Programul ar fi putut folosi, pur și simplu, clasa *dogs*, la fiecare creare a unui obiect de tipul câine fără pete. Totuși, pentru îmbunătățirea clarității programului (pentru a permite programului să lucreze cu câinii cu pete și câinii fără pete, în loc de câini cu pete și câini), s-a derivat clasa *unspotted\_dogs*.





## SĂ ÎNȚELEGEM MOȘTENIREA CLASELOR

Când se definesc clase într-un program, apar situații când două sau mai multe clase au caracteristici comune (aceiași membri). În loc de a reproduce membrii în fiecare clasă, se poate defini o *clasă de bază* ce conține funcțiile membru comune. Apoi, se pot construi (deriva) clase succesive din clasa de bază. În acest caz, clasele *derivate* moștenesc caracteristicile clasei de bază. Să presupunem, de exemplu, că definiți următoarea clasă *vehicle*:

```
class vehicle {
public:
    vehicle(char *name, int wheels, int engine);
    void show_vehicle(void);
private:
    char name[64];

    int wheels;
    int engine;
};
```

Folosind moștenirea, se poate declara o clasă *motorcycle* astfel:

```
class motorcycle : public vehicle {
public:
    motorcycle(char *name, int wheels, int engine, int seats);
    void show_cycle(void);
private:
    int seats;
};
```

În același mod, se poate declara și clasa *automobile*:

```
class automobile: public vehicle {
public:
    automobile(char *name, int wheels, int engine, int doors);
    void show_auto(void);
private:
    int doors;
};
```

În acest caz, clasele *motorcycle* și *automobile* moștenesc amândouă caracteristicile definite în clasa *vehicle*.

## FOLOSIREA FUNCȚIILOR CONSTRUCTOR LA MOȘTENIREA CLASELOR

Așa cum ați învățat în Capitolul 2, funcțiile constructor permit programelor să inițializeze variabilele membru ale claselor. Tot constructorii se folosesc pentru inițializări și în cazul moștenirii claselor. Singura diferență este aceea că, constructorul clasei derivate trebuie să apeleze constructorul clasei de bază. Următoarea funcție implementează funcția constructor a clasei de bază *dogs*:

```
dogs::dogs(char *breed, int height, int weight, char *color)
{
    strcpy(dogs::breed, breed);
    dogs::height = height;
    dogs::weight = weight;
    strcpy(dogs::color, color);
}
```

Funcțiile constructor ale claselor derivate folosesc o sintaxă similară definiției clasei, având simbolul ":" urmat de funcția constructor a clasei de bază:

```

    Constructor clasă derivată
spotted dogs::spotted dogs(char *breed, int height, int weight,
    char *color, char *spot_color) : dogs (breed, height, weight,
    color)
{
    Constructor clasă de bază
    strcpy(spotted dogs::spot_color, spot_color);
}

unspotted dogs::unspotted dogs(char *breed, int height, int weight,
    char *color) : dogs(breed, height, weight, color)
{
    // Do nothing-base class constructor initialized members
}
```

Așa cum se poate vedea, apelul constructorului clasei de bază folosește nume de parametri identici celor transferați constructorului clasei derivate.

```

    Parametrii clasei derivate
spotted dogs::spotted dogs(char *breed, int height, int weight,
    char *color, char *spot_color) : dogs (breed, height, weight,
    color)
{
    Parametrii clasei de bază
    strcpy(spotted dogs::spot_color, spot_color);
}

unspotted dogs::unspotted dogs(char *breed, int height, int weight,
    char *color) : dogs(breed, height, weight, color)
{
    // Do nothing-base class constructor initialized members
}
```

Când programul apelează constructorul clasei *spotted\_dogs*, C++ va apela automat, mai întâi, constructorul clasei *dogs*. Funcția constructor a clasei de bază este întotdeauna apelată înaintea constructorului clasei derivate. În acest mod, membrii clasei de bază sunt corect inițializați, iar memoria va fi corect alocată pentru buffer-ele clasei de bază, înainte de referirea clasei derivate.

Dacă examinați funcția constructor a clasei *spotted\_dogs*, veți vedea că inițializează variabila membru *spot\_color*. În schimb, constructorul *unspotted\_dogs* nu are valori de atribuit (în afara celor folosite de constructorul clasei de bază).

Următorul program, NEW\_DOGS.CPP folosește moștenirea pentru a crea clasele de câini cu pete și fără pete.

```
#include <iostream.h>
#include <string.h>

class dogs {
public:
    dogs(char *breed, int height, int weight, char *color);
    void show_breed(void);
private:
    char breed[64];
    int height;
    int weight;
    char color[64];
};

class spotted_dogs : public dogs {
public:
    spotted_dogs(char *breed, int height, int weight, char *color,
        char *spot_color);
    void show_breed(void);
    void spot_info(void);
private:
    char spot_color[64];
};

class unspotted_dogs : public dogs {
public:
    unspotted_dogs(char *breed, int height, int weight,
        char *color);
};

dogs::dogs(char *breed, int height, int weight, char *color)
{
    strcpy(dogs::breed, breed);
    dogs::height = height;
    dogs::weight = weight;
    strcpy(dogs::color, color);
}
```

```
spotted_dogs::spotted_dogs(char *breed, int height, int weight,
    char *color, char *spot_color) : dogs (breed, height, weight,
    color)
{
    strcpy(spotted_dogs::spot_color, spot_color);
}

unspotted_dogs::unspotted_dogs(char *breed, int height, int weight,
    char *color) : dogs(breed, height, weight, color)
{
    // Do nothing-base class constructor initialized members
}

void dogs::show_breed(void)
{
    cout << "Breed: " << breed << endl;
    cout << "Height: " << height << " Weight: " << weight << endl;
    cout << "Color: " << color << endl;
}

void spotted_dogs::show_breed(void)
{
    dogs::show_breed();
    cout << "Spot color: " << spot_color << endl << endl;
}

void spotted_dogs::spot_info(void)
{
    cout << "This breed has " << spot_color << " spots" << endl <<
        endl;
}

void main(void)
{
    spotted_dogs happy("Dalmatian", 24, 60, "white",
        "black or brown (liver)");
    unspotted_dogs rover("Labrador Retriever", 24, 65,
        "black or yellow");

    happy.show_breed();
    happy.spot_info();

    rover.show_breed();
}
```

Deoarece clasa *spotted\_dogs* conține membrul *spot\_color*, clasa folosește funcția membru *show\_breed* proprie. Folosind operatorul de rezoluție globală (::) funcția apelează mai întâi funcția *show\_breed* a clasei de bază pentru a afișa variabilele membru comune tuturor raselor de câini. Apoi, funcția afișează valoarea propriei variabile membru:

```
void spotted_dogs::show_breed(void)
{
    dogs::show_breed();
    cout << "Spot color: " << spot_color << endl << endl;
}
```

Apelarea funcției clasei de bază

Afișarea variabilei membru `spot_color`

La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> NEW DOGS <ENTER>
Breed: Dalmatian
Height: 24 Weight: 60
Color: white
Spot color: black or brown (liver).
This breed has black or brown (liver) spots
```

```
Breed: Labrador Retriever
Height: 24 Weight: 65
Color: black or yellow
```

Să observăm că în ieșirea programului s-a schimbat următoarea linie:

```
Dalmatian has black or brown (liver) spots
This breed has black or brown (liver) spots
```

Programul folosește funcția `spot_info` pentru a afișa mesajul corespunzător:

```
void spotted_dogs::spot_info(void)
{
    cout << "This breed has " << spot_color << " spots" << endl << endl;
}
```

În acest caz, funcția nu poate folosi variabila `breed` a clasei `dogs`, care conține numele rasei (Dalmatian), deoarece aceasta nu este publică în clasa `dogs`. Singurul mod prin care o clasă derivată poate avea acces la membrii privați ai clasei de bază este de a folosi funcțiile membru ale clasei de bază, cum este `show_breed`. Un alt mod de a rezolva această problemă este de a face variabila membru `breed` publică în clasa `dogs`. De asemenea, așa cum veți vedea mai târziu, se poate declara variabila ca fiind un membru protejat (*protected*) al clasei, ceea ce va permite accesul direct numai prin intermediul obiectelor clasei derivate sau ale clasei de bază.

#### ANALIZA UNUI EXEMPLU

Pentru a înțelege mai bine cum și când sunt apelate funcțiile constructor și destructor în situația moștenirii claselor, următorul program, `GENERIC.CPP`, definește două clase, denumite `base` și `derived`. În cadrul fiecărui constructor sau destructor, programul afișează un mesaj ce identifică funcția curentă:

```
#include <iostream.h>
#include <string.h>
```

```
class base {
public:
    base(char *base_message);
    ~base(void);
private:
    char base_message[64];
};

class derived : public base {
public:
    derived(char *derived_message);
    ~derived(void);
private:
    char derived_message[64];
};

base::base(char *message)
{
    strcpy(base_message, message);
    cout << "In base class constructor: " << message << endl;
}

base::~base(void)
{
    cout << "In base class destructor: " << base_message << endl;
}

derived::derived(char *message) : base("Hello, base")
{
    strcpy(derived_message, message);
    cout << "In derived class constructor: " << message << endl;
}


derived::~~derived(void)
{
    cout << "In derived class destructor: " << derived_message << endl;
}

void main(void)
{
    derived object("Hello, world");
}
```

La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> GENERIC <ENTER>
In base class constructor: Hello, base
In derived class constructor: Hello, world
In derived class destructor: Hello, world
In base class destructor: Hello, base
```

După cum se vede, când programul instanțiază obiectul derivat, este apelată mai întâi funcția clasei de bază, urmată de constructorul clasei derivate. La distrugerea obiectului, secvența de apel se desfășoară invers, fiind apelat mai întâi destructorul clasei derivate, apoi destructorul clasei de bază.



### SĂ ÎNȚELEM CUVÂNTUL – CHEIE PUBLIC

Când se creează o clasă derivată, se specifică numele acesteia, urmat de simbolul ":" și de numele clasei de bază, ca mai jos:

```
class derived : public base {
    // Membrii
};
```

După cum se observă, definiția clasei derivate folosește cuvântul-cheie *public*. Aceasta înseamnă că membrii de tip *public* ai clasei de bază sunt considerați de același tip și în clasa derivată. În același mod, membrii protejați (*protected*) ai clasei de bază sunt tratați ca membri protejați în clasa derivată. Majoritatea programelor vor folosi cuvântul-cheie *public* ca mai sus. Dacă la definiția unei clase se folosește cuvântul-cheie *private*, atunci membrii publici și protejați ai clasei de bază vor fi tratați drept membri nonpublici (*private*) în clasa derivată.

#### ANALIZA ALTUI EXEMPLU

Când vă gândiți la o unitate de disc, îi asociați întotdeauna anumite atribute, precum: capacitatea de memorare, geometria discului (numărul de fețe, piste, sectoare etc.), sau viteza de transfer. Așa cum se arată în Figura 4.1, aceste caracteristici sunt moștenite atât de hard-disk-uri cât și de dischete, fiecare dintre ele având și caracteristici proprii, ca tipul de interfață (SCSI sau IDE) sau mecanismul de protecție la scriere.

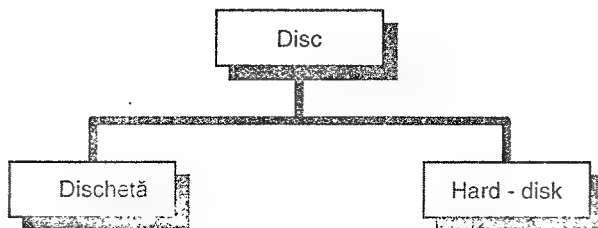


Figura 4.1. Relația dintre tipurile de disc

Următorul program, DISKCLAS.CPP, creează o clasă de bază numită *disk*:

```

class disk {
public:
    disk(char *name, int sides, int tracks, int sectors_per_track,
        int bytes_per_sector);
    void show_disk(void);
private:
    char name[64];
    int sides;
    int tracks;
    int sectors_per_track;
    int bytes_per_sector;
    long capacity;
};
    
```

Apoi, programul derivează clasele *hard\_disk* și *floppy\_disk*, ca mai jos:

```

class floppy_disk : public disk {
public:
    floppy_disk(char *name, int sides, int tracks,
        int sectors_per_track, int bytes_per_sector, int state);
    void set_write_protect(int state);
    void show_floppy(void);
private:
    int write_protect_state;
};

class hard_disk : public disk {
public:
    hard_disk(char *name, int sides, int tracks,
        int sectors_per_track, int bytes_per_sector,
        char *controller_type);
    void show_hard_disk(void);
private:
    char controller_type[64];
};
    
```

Programul DISKCLAS.CPP este redat în întregime în continuare:

```

#include <iostream.h>
#include <string.h>

class disk {
public:
    disk(char *name, int sides, int tracks, int sectors_per_track,
        int bytes_per_sector);
    void show_disk(void);
private:
    char name[64];
    int sides;
    int tracks;
    int sectors_per_track;
};
    
```

```

    int bytes_per_sector;
    long capacity;
};

class floppy_disk : public disk {
public:
    floppy_disk(char *name, int sides, int tracks,
        int sectors_per_track, int bytes_per_sector, int state);
    void set_write_protect(int state);
    void show_floppy(void);
private:
    int write_protect_state;
};

class hard_disk : public disk {
public:
    hard_disk(char *name, int sides, int tracks,
        int sectors_per_track, int bytes_per_sector,
        char *controller_type);
    void show_hard_disk(void);
private:
    char controller_type[64];
};

disk::disk(char *name, int sides, int tracks, int sectors_per_track,
    int bytes_per_sector)
{
    strcpy(disk::name, name);
    disk::sides = sides;
    disk::tracks = tracks;
    disk::sectors_per_track = sectors_per_track;
    disk::bytes_per_sector = bytes_per_sector;
    capacity = (long) sides * (long) tracks;
    capacity *= (long) sectors_per_track * (long) bytes_per_sector;
}

void disk::show_disk(void)
{
    cout << "Disk name: " << name << endl;
    cout << "Sides: " << sides << " Tracks: " << tracks << endl;
    cout << "Sectors per track: " << sectors_per_track << endl;
    cout << "Bytes per sector: " << bytes_per_sector << endl;
    cout << "Capacity: " << capacity << " bytes" << endl;
}

floppy_disk::floppy_disk(char *name, int sides, int tracks,
    int sectors_per_track, int bytes_per_sector, int state) :
    disk(name, sides, tracks, sectors_per_track, bytes_per_sector)

```

```

{
    write_protect_state = state;
}

void floppy_disk::show_floppy(void)
{
    show_disk();
    cout << "Write-protect is: " << ((write_protect_state) ?
        "On": "Off") << endl << endl;
}

hard_disk::hard_disk(char *name, int sides, int tracks,
    int sectors_per_track, int bytes_per_sector, char *controller) :
    disk(name, sides, tracks, sectors_per_track, bytes_per_sector)
{
    strcpy(controller_type, controller);
}

void hard_disk::show_hard_disk(void)
{
    show_disk();
    cout << "Controller type: " << controller_type << endl << endl;
}

void main(void)
{
    hard_disk ide_drive("Hard disk", 4, 615, 80, 512, "IDE");
    floppy_disk high_density("Floppy disk", 2, 80, 18, 512, 1);

    ide_drive.show_hard_disk();
    high_density.show_floppy();
}

```

La compilarea și execuția acestui program, pe ecran va apărea:

```

C:\> DISKCLAS <ENTER>
Disk name: Hard disk
Sides: 4 Tracks: 615
Sectors per track: 80
Bytes per sector: 512
Capacity: 100761600 bytes
Controller type: IDE

```

```

Disk name: Floppy disk
Sides: 2 Tracks: 80
Sectors per track: 18
Bytes per sector: 512
Capacity: 1474560 bytes
Write-protect is: On

```

## UN ULTIM EXEMPLU DE MOȘTENIRE PE UN SINGUR NIVEL

Probabil că deja v-ați obișnuit cu sintaxa claselor de bază și a celor derivate. Sintaxa reprezintă "știința" programării orientate pe obiecte. "Arta" acestei programări începe când trebuie să determinăm care din atribute aparțin claselor de bază, și care atribute trebuie să figureze în clasele derivate. Cea mai bună metodă de a stăpâni "arta" proiectării orientate pe obiect este de a exersa în permanență, de a rafina treptat conceptele introduse. Să analizăm din nou un exemplu. Să presupunem că trebuie să creăm o bază de date a angajaților unei mari companii. Compania are manageri cu salariu anual, angajați permanenți plătiți cu ora, și angajați temporari plătiți cu ora. Pentru fiecare tip de angajat, programul trebuie să urmărească următoarele informații:

Salariații	Permanent cu ora	Temporar cu ora
nume	nume	nume
telefon acasă	telefon acasă	telefon acasă
telefon serviciu	telefon serviciu	telefon serviciu
salariu anual	plata pe oră	plata pe oră
bonificații		
superior ierarhic	superior ierarhic	superior ierarhic
asistent		

Prima "mișcare" care se face la proiectarea clasei de bază este de a selecta atributele comune, ca mai jos:

Salariații	Permanenții cu ora	Temporarii cu ora
nume	nume	nume
telefon acasă	telefon acasă	telefon acasă
telefon serviciu	telefon serviciu	telefon serviciu
salariu anual	plata pe oră	plata pe oră
bonificații		
superior ierarhic	superior ierarhic	superior ierarhic
asistent		

Folosind atributele comune, se poate acum crea clasa de bază *employee*, cum se arată mai jos:

```
class employee {
public:
    employee(char *name, char *home_phone, char *office_phone,
```

```
    char *reports_to);
    void show_employee(void);
private:
    char name[64];
    char home_phone[64];
    char office_phone[64];
    char reports_to[64];
};
```

Apoi, înlăturând atributele comune, clasele derivate devin:

```
class salaried : public employee {
public:
    salaried(char *name, char *home_phone, char *office_phone,
        char *reports_to, float salary, float bonus_level,
        char *assistant);
    void show_salaried(void);
private:
    float salary;
    float bonus_level;
    char assistant[64];
};

class hourly : public employee {
public:
    hourly(char *name, char *home_phone, char *office_phone,
        char *reports_to, float wage);
    void show_hourly(void);
private:
    float wage;
};

class temporary : public employee {
public:
    temporary(char *name, char *home_phone, char *office_phone,
        char *reports_to, float wage);
    void show_temporary(void);
private:
    float wage;
};
```

Dacă examinați clasele *hourly* și *temporary*, veți vedea că membrii lor sunt identici. Tocmai aici arta, preferințele programatorului și alți asemenea factori intră în rol.

În funcție de modul în care programul lucrează cu obiectele, se poate crea o variabilă membru în clasa *hourly* cu rol de indicator, pentru a preciza dacă un obiect corespunde unui angajat permanent sau temporar, ca mai jos:

```
enum worker_type { permanent, temporary };
```

```
class hourly : public employee {
```

```
public:
    temporary(char *name, char *home_phone, char *office_phone,
        char *reports_to, float wage, worker_type flag);
    void show_temporary(void);
private:
    float wage;
    worker_type flag;
};
```

În cadrul funcțiilor membru ale clasei, codul acestora trebuie să examineze variabila indicator pentru a determina tipul corect.

Există unele situații în care se dorește restrângerea anumitor operații din program la angajații temporari sau permanenți. În astfel de situații obiectele tip angajat nu mai sunt aceleași, iar programul trebuie să folosească două tipuri distincte de clase. În acest fel, claritatea programului se va îmbunătăți. De exemplu, următoarele fragmente ilustrează o funcție care limitează operațiile numai asupra angajaților temporari cu ora:

```
float temporary::pay_overtime(void)    float hourly::pay_overtime(void)
{
    // Statements
}
{
    if (flag == temporary)
    {
        // Statements
    }
}
```

Prin simpla examinare a antetului funcției *pay-overtime* din partea stângă, o persoană care citește programul poate imediat să determine că funcția se aplică numai obiectelor de tip angajat temporar. În cazul funcției din partea dreaptă, cititorul programului trebuie să aibă acces la instrucțiunile sursă (în particular, instrucțiunea *if*) pentru a afla că funcția afectează numai obiectele de tip permanent.

Următorul program, WORKERS.CPP, folosește clasele *employee*, *salaried*, *permanent* și *temporary*:

```
#include <iostream.h>
#include <string.h>

class employee {
public:
    employee(char *name, char *home_phone, char *office_phone,
        char *reports_to);
    void show_employee(void);
private:
    char name[64];
    char home_phone[64];
    char office_phone[64];
```

```
char reports_to[64];
};

class salaried : public employee {
public:
    salaried(char *name, char *home_phone, char *office_phone,
        char *reports_to, float salary, float bonus_level,
        char *assistant);
    void show_salaried(void);
private:
    float salary;
    float bonus_level;
    char assistant[64];
};

class hourly : public employee {
public:
    hourly(char *name, char *home_phone, char *office_phone,
        char *reports_to, float wage);
    void show_hourly(void);
private:
    float wage;
};

class temporary : public employee {
public:
    temporary(char *name, char *home_phone, char *office_phone,
        char *reports_to, float wage);
    void show_temporary(void);
private:
    float wage;
};

employee::employee(char *name, char *home_phone, char *office_phone,
    char *reports_to)
{
    strcpy(employee::name, name);
    strcpy(employee::home_phone, home_phone);
    strcpy(employee::office_phone, office_phone);
    strcpy(employee::reports_to, reports_to);
}

void employee::show_employee(void)
{
    cout << "Name: " << name << endl;
    cout << "Home phone: " << home_phone << endl;
    cout << "Office phone: " << office_phone << endl;
    cout << "Reports to: " << reports_to << endl;
}
```



```

salaried::salaried(char *name, char *home_phone, char *office_phone,
    char *reports_to, float salary, float bonus_level,
    char *assistant) : employee(name, home_phone, office_phone,
    reports_to)
{
    salaried::salary = salary;
    salaried::bonus_level = bonus_level;
    strcpy(salaried::assistant, assistant);
}

void salaried::show_salaried(void)
{
    show_employee();
    cout << "Salary: $" << salary << endl;
    cout << "Bonus level: $" << bonus_level << endl;
    cout << "Asistent: " << assistant << endl;
}

hourly::hourly(char *name, char *home_phone, char *office_phone,
    char *reports_to, float wage) : employee(name, home_phone,
    office_phone, reports_to)
{
    hourly::wage = wage;
}

void hourly::show_hourly(void)
{
    show_employee();
    cout << "Wage: $" << wage << endl;
}

temporary::temporary(char *name, char *home_phone,
    char *office_phone, char *reports_to, float wage) :
    employee(name, home_phone, office_phone, reports_to)
{
    temporary::wage = wage;
}

void temporary::show_temporary(void)
{
    show_employee();
    cout << "Wage: $" << wage << endl;
}

void main(void)
{
    salaried top_boss("Joe Smith", "555-1111", "555-1112",
        "Mark Jones", 30000.0, 10000.0, "Alicia Jones");

    hourly typist("David Kline", "555-2222", "555-2223",
        "John Martin", 4.50);

```

```

temporary receptionist("Mary Scott", "555-3333", "555-3334",
    "Doris Davis", 4.00);

top_boss.show_salaried();

cout << endl << endl;

typist.show_hourly();

cout << endl << endl;

receptionist.show_temporary();
}

```

La compilarea și execuția acestui program, pe ecran vor apărea următoarele rezultate:

```

C:\> WORKERS <ENTER>
Name: Joe Smith
Home phone: 555-1111
Office phone: 555-1112
Reports to: Mark Jones
Salary: $30000
Bonus level: $10000
Assistant: Alicia Jones

```

```

Name: David Kline
Home phone: 555-2222
Office phone: 555-2223
Reports to: John Martin
Wage: $4.5

```

```

Name: Mary Scott
Home phone: 555-3333
Office phone: 555-3334
Reports to: Doris Davis
Wage: $4

```

#### CHEIA SUCCESULUI



#### PROTOTIPURI, REVIZII, FINALIZĂRI

Programarea orientată pe obiecte este atât artă, cât și știință. După ce ați scris primele programe, știința programării orientate pe obiecte devine destul de clară. Artă proiectării programelor constă în atribuirea corectă a membrilor către clasele de bază sau derivate.

De regulă, se consideră prima clasă proiectată și membrii ei drept un prototip. Pe măsură ce dezvoltați programul, revedeți și actualizați membrii claselor. În majoritatea cazurilor, aceasta duce la eliminarea rescrierii codului într-o etapă ulterioară, cauzată de o proiectare inițială incorectă.

În general, dacă vă găsiți în situația de a transmite parametri unor funcții ce conțin informații suplimentare despre un obiect, atunci trebuie să refaceți toată proiectarea. Dacă în schimb extindeți codul, atunci rămâne o ușă deschisă pentru erori, iar claritatea programului va avea de suferit.

### SĂ ÎNȚELEGEM MOȘTENIREA MULTIPLĂ

Există situații în care unele clase folosesc caracteristicile a două sau a mai multor clase de bază. Moștenirea multiplă este folosirea a două sau mai multor clase de bază pentru derivarea altei clase. Să presupunem, de exemplu, că s-au declarat clasele *book* și *disk*:

```
class book {
public:
    book(char *title, char *author, int pages);
    void show_book(void);
private:
    char title[64];
    char author[64];
    int pages;
};

class disk {
public:
    disk(float capacity);
    void show_disk(void);
private:
    float capacity;
};

class bundle : public book, public disk {
public:
    bundle(char *title, char *author, int pages, float capacity,
           float price);
    void show_bundle(void);
private:
    float price;
};
```

Se poate apoi crea o clasă denumită *bundle* care este o combinație între cele două clase anterioare:

```
class bundle : public book, public disk {
public:
    bundle(char *title, char *author, int pages, float capacity,
           float price);
    void show_bundle(void);
};
```

```
private:
    float price;
```

În acest caz, *bundle* este clasa derivată, iar *disk* și *book* sunt clasele de bază:

```
class bundle : public book, public disk {
```

Clasă derivată

Clase de bază

Când o clasă este derivată din două sau mai multe clase, clasele de la bază se separă prin virgule, cum s-a arătat mai sus. La declararea funcției constructor al clasei, trebuie specificat constructorul clasei de bază într-un mod similar:

```
bundle::bundle(char *title, char *author, int pages, float capacity,
               float price) : book(title, author, pages), disk(capacity)
{
    bundle::price = price;
}
```

Constructor clasă derivată

Constructor clasă de bază

În acest caz, C++ va apela mai întâi constructorul *book*, urmat de funcția *disk* și în final de funcția *bundle*.

Următorul program, `MULT_INH.CPP`, ilustrează moștenirea multiplă prin crearea clasei *bundle*. Programul nu face altceva decât să apeleze funcțiile constructor și destructor, care la rândul lor afișează mesaje pentru urmărirea execuției:

```
#include <iostream.h>
#include <string.h>

class book {
public:
    book(char *title, char *author, int pages);
    void show_book(void);
private:
    char title[64];
    char author[64];
    int pages;
};

class disk {
public:
    disk(float capacity);
    void show_disk(void);
private:
    float capacity;
};

class bundle : public book, public disk {
```

```

public:
    bundle(char *title, char *author, int pages, float capacity,
           float price);
    void show bundle(void);
private:
    float price;
};

book::book(char *title, char *author, int pages)
{
    strcpy(book::title, title);
    strcpy(book::author, author);
    book::pages = pages;
}

void book::show book(void)
{
    cout << "Title: " << title << endl;
    cout << "Author: " << author << endl;
    cout << "Pages: " << pages << endl;
}

disk::disk(float capacity)
{
    disk::capacity = capacity;
}

void disk::show disk(void)
{
    cout << "Capacity: " << capacity << "Mb" << endl;
}

bundle::bundle(char *title, char *author, int pages, float capacity,
               float price) : book(title, author, pages), disk(capacity)
{
    bundle::price = price;
}

void bundle::show bundle(void)
{
    show book();
    show disk();
    cout << "Price: $" << price << endl;
}

void main(void)
{
    bundle this book("Jamsa's 1001 C/C++ Tips", "Jamsa", 896,
                    1.44, 39.95);

    this book.show bundle();
}

```

La compilarea și execuția acestui program, pe ecran va apărea:

```

C:\> MULT_INH <ENTER>
Title: Jamsa's 1001 C/C++ Tips
Author: Jamsa
Pages: 896
Capacity: 1.44Mb
Price: $39.95

```

După cum se poate vedea, programul apelează funcțiile constructor ale claselor *book*, *disk* și *bundle*, în această ordine.

#### CHEIA SUCCESULUI



#### SĂ ÎNȚELEM MOȘTENIREA MULTIPLĂ

Moștenirea multiplă constă în folosirea a două sau a mai multor clase de bază entru derivarea unei clase noi. La derivarea noii clase din clasele de bază, trebuie specificat numele clasei derivate, urmat de numele claselor de bază, cum se arată mai jos:

```

class derived : public base1, public base2;
{
    // members
}

```

Fiecare din clasele de bază este precedată de cuvântul-cheie *public*. Acesta permite ca în clasele derivate, membrii *public* și *protected* ai claselor de bază să fie tratați respectiv ca membri *public* și *protected* ai clasei derivate. La definirea constructorului clasei derivate se specifică și constructorul clasei de bază, ca în exemplu următor:

```

derived::derived(int a, int b): base1(a), base2(b)
{
    // Statements
}

```

În acest caz, când programul va apela constructorul clasei *derived*, C++ va apela mai întâi constructorul clasei *base1*, apoi constructorul clasei *base2*.

#### SĂ ÎNȚELEM MOȘTENIREA PE MAI MULTE NIVELE

Așa cum am văzut, moștenirea multiplă constă în folosirea a două sau mai multor clase la bază pentru a deriva o nouă clasă. **Moștenirea pe mai multe nivele**, pe de altă parte, are loc la derivarea unei clase dintr-o clasă de bază care, la rândul ei, este derivată din altă clasă. De exemplu, Figura 4.2 ilustrează clasa *manager*, bazată pe clasa *worker*, aceasta fiind la rândul ei bazată pe clasa *person*.

Pentru a crea cele 3 clase, începem cu clasa *person*, ca mai jos:

```
class person {
public:
    person(char *name, int age);
    void show_person(void);
private:
    char name[64];
    int age;
};
```

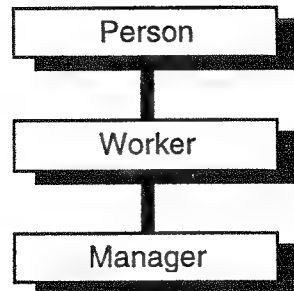


Figura 4.2. O moștenire pe nivele multiple.

Apoi, definim clasa *worker* astfel:

```
class worker : public person {
public:
    worker(char *name, int age, char *phone, float wage);
    void show_worker(void);
private:
    char phone[64];
    float wage;
};
```

În sfârșit, vom defini clasa *manager*:

```
class manager : public worker {
public:
    manager(char *name, int age, char *phone, float wage,
            char *office);
    void show_manager(void);
private:
    char office[64];
};
```

Următorul program, MULTILVL.CPP ilustrează moștenirea pe nivele multiple:

```
#include <iostream.h>
#include <string.h>
```

```
class person {
public:
```

```
    person(char *name, int age);
    void show_person(void);
private:
    char name[64];
    int age;
};

class worker : public person {
public:
    worker(char *name, int age, char *phone, float wage);
    void show_worker(void);
private:
    char phone[64];
    float wage;
};

class manager : public worker {
public:
    manager(char *name, int age, char *phone, float wage,
            char *office);
    void show_manager(void);
private:
    char office[64];
};

person::person(char *name, int age)
{
    strcpy(person::name, name);
    person::age = age;
}

void person::show_person(void)
{
    cout << endl << "Name: " << name << endl;
    cout << "Age: " << age << endl;
}

worker::worker(char *name, int age, char *phone, float wage) :
    person(name, age)
{
    strcpy(worker::phone, phone);
    worker::wage = wage;
}

void worker::show_worker(void)
{
    show_person();
    cout << "Phone: " << phone << endl;
}

manager::manager(char *name, int age, char *phone, float wage,
                char *office) : worker(name, age, phone, wage)
```

```
{
    strcpy(manager::office, office);
}

void manager::show_manager(void)
{
    show_worker();
    cout << "Office: " << office << endl;
}

void main(void)
{
    worker security("Ken Smith", 43, "555-1212", 4.50);

    manager boss("Betty Louis", 30, "555-2121", 12.50, "Room 3B");

    security.show_worker();

    boss.show_manager();
}
```

La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> MULTILVL <ENTER>
Name: Ken Smith
Age: 43
Phone: 555-1212
```

```
Name: Betty Louis
Age: 30
Phone: 555-2121
Office: Room 3B
```

### SĂ ÎNȚELEM MEMBRII DE TIP PROTECTED AI UNEI CLASE

Așa cum ați învățat în Capitolul 2, funcțiile membru ale clasei se pot declara de tip *public* sau *private*. La declararea unui membru ca *public*, acesta va fi accesibil în tot programul (când obiectul este în domeniu - vezi Capitolul 9). Un membru de tipul *private* poate fi accesibil numai prin funcțiile membru ale clasei. În programele din acest capitol, clasele derivate au avut acces numai la membrii de tip *public* ai clasei de bază. De exemplu, următorul program, INPUBLIC.CPP, ilustrează modul în care o clasă derivată poate avea acces la membrii de tip *public* ai clasei de bază. În acest caz, programul are acces la membrul de tip *public* *base\_number* al clasei de bază, dar nu are acces la membrul nonpublic *base\_message*:

```
#include <iostream.h>
#include <string.h>
```

```
class base {
public:
    base(char *base_message, int number);
    int base_number;
private:
    char base_message[64];
};

class derived : public base {
public:
    derived(char *derived_message, int number);
private:
    char derived_message[64];
};

base::base(char *message, int number)
{
    strcpy(base_message, message);
    base_number = number;
}

derived::derived(char *message, int number) : base("Base message",
    number)
{
    strcpy(derived_message, message);
}

void main(void)
{
    derived object("Hello, world", 1001);

    cout << "The base number is " << object.base_number << endl;
}
```

La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> INPUBLIC <ENTER>
The base number is 1001
```

Exersați cu acest program astfel încât funcția *main* să facă referință la variabila membru *base\_message*, în loc de *base\_number*. Deoarece membrii nonpublici nu sunt accesibili în afara clasei, nici obiectele derivate nu vor avea acces la aceștia.

În funcție de intenția programului, pot apărea situații când se dorește ca unii membri ai clasei de bază să poată fi referiți de obiectele clasei derivate, fiind totuși protejați de restul programului. În aceste cazuri se pot folosi în program membri protejați. Un membru protejat (*protected*) al unei clase poate fi direct folosit de clasele derivate, dar nu și de alte părți ale programului. Următorul program, PROTECT.CPP, ilustrează folosirea membrilor protejați ai unei clase:

```
#include <iostream.h>
#include <string.h>

class base {
public:
    base(char *base_message, int number);
protected:
    int base_number;
private:
    char base_message[64];
};

class derived : public base {
public:
    derived(char *derived_message, int number);
    void show_number(void);
private:
    char derived_message[64];
};

base::base(char *message, int number)
{
    strcpy(base_message, message);
    base_number = number;
}

derived::derived(char *message, int number) : base("Base message",
number)
{
    strcpy(derived_message, message);
}

void derived::show_number(void)
{
    cout << "The base-class number is " << base_number << endl;
}

void main(void)
{
    derived object("Hello, world", 1001 );

    object.show_number();
}
```

La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> PROTECT <ENTER>
The base-class number is 1001
```

Exersați acest program. De exemplu, încercați să adăugați următoarele instrucțiuni la funcția *main*:

```
void main(void)
{
    derived object("Hello, world", 1001 );

    object.show_number();

    cout << "The base-class number is " << base_number << endl;
}
```

Deoarece programul nu are acces la membrii protejați ai clasei, compilatorul va genera erori de sintaxă.

#### CHEIA SUCCESULUI



#### SĂ ÎNȚELEM MEMBRII PROTEJAȚI AI UNEI CLASE

Clasele derivate nu au acces la membrii de tip *private* ai clasei de bază. Există situații când clasele derivate trebuie să primească drepturi speciale de acces la membrii clasei de bază. Atunci se pot folosi membrii protejați ai clasei de bază, care sunt accesibili de către clasele derivate, dar nu și din alte părți ale programului. De exemplu, dată fiind următoarea definiție de clasă, clasele derivate au acces la membrii *filename* și *size*, dar nu și la variabila membru *some\_value* de tip *private*:

```
class base {
public:
    base(char name[64], int value);
    void show_base(void);
protected:
    char filename[64];
    long size;
private:
    int some_value;
};
```

#### REZUMAT

Moștenirea constă în folosirea unei clase de bază pentru a deriva o nouă clasă. Moștenirea multiplă constă în folosirea a două sau mai multor clase pentru a deriva o altă clasă. Așa cum ați învățat în acest capitol, C++ suportă complet atât moștenirea simplă, cât și moștenirea multiplă. Folosind avantajul moștenirii, se poate economisi cod de program prin exploatarea relațiilor între obiecte. Moștenirea este un concept cheie folosit în proiectarea și programarea orientată pe obiecte. Înainte de a trece la Capitolul 5, asigurați-vă că ați învățat următoarele:

- ✓ Moștenirea are loc atunci când o clasă derivată partajează (moștenește) caracteristici (membri) similare cu clasa de bază. Aceasta vă permite să economisiți timp și programe.
- ✓ La folosirea moștenirii, se pot utiliza în continuare constructori la inițIALIZAREA claselor, dar constructorul clasei derivate trebuie să apeleze constructorul clasei de bază. Funcțiile constructor al clasei derivate folosesc o sintaxă similară definiției clasei, cu simbolul „:” urmat de constructorul clasei de bază.
- ✓ Sintaxa este „știința” programării orientată pe obiecte. „Arta” acestui tip de programare începe atunci când trebuie să determinați atributele ce aparțin clasei de bază și cele care aparțin claselor derivate.
- ✓ La proiectarea claselor, pot apărea situații când anumite clase folosesc caracteristicile a două sau mai multor clase de bază. Aceasta este moștenirea multiplă.
- ✓ Când o clasă este derivată din două sau mai multe clase, clasele de bază se separă prin virgule.
- ✓ Moștenirea pe mai multe nivele are loc când o clasă este derivată dintr-o clasă de bază, care, la rândul ei, este derivată din altă clasă.
- ✓ Un membru proiectat al unei clase poate fi direct accesibil de clasele derivate, dar imposibil de apelat din altă parte a programului. Acest tip de membru se folosește pentru a oferi membrilor derivați acces direct la membrii clasei de bază, în același timp protejându-i pe aceștia din urmă de restul programului.

## CAPITOLUL 5

### ACOMODAREA CU SUPRAPUNEREA FUNCȚIILOR ȘI OPERATORILOR

Prin crearea claselor în programele C++, se obțin propriile tipuri de date. În sensul cel mai simplu, un *tip* definește un set de valori de date ce pot fi memorate și un set de operații ce se pot efectua cu aceste date. De exemplu, o variabilă de tip *int* poate memora valori întregi în domeniul -32 768, 32 767. Operațiile ce pot fi realizate cu această variabilă sunt adunarea, scăderea, înmulțirea, împărțirea și mai multe operații bit cu bit.

La crearea claselor, este de dorit ca unele operații să fie exprimate prin operatori. De exemplu, să presupunem că într-un program se folosește o variabilă a unei clase ce exprimă data sub forma zi, lună, an. Dacă se dorește adunarea a 30 de zile la dată, sau scăderea a 15 zile din dată, operatorii plus și minus pot produce un cod foarte inteligibil, de forma:

```
invoice_date = order_date + 30;
```

```
first_notice = invoice_date - 15;
```

Aveți în vedere, totuși, că variabilele membru ale clasei conțin câmpurile zi, lună și an. Pentru ca operatorii plus și minus să aibă sens, programul trebuie să definească operațiile ce vor trebui efectuate la întâlnirea acestor operatori. Cu alte cuvinte, programul va efectua un anumit set de operații (probabil adunarea și scăderea) când va întâlni operatorii plus și minus folosiți cu variabile de tip *int* sau *float*, și alt set de operații când va întâlni aceiași operatori folosiți cu variabilele clasei de date. **Suprapunerea operatorilor** este procesul de atribuire a două sau a mai multor operații aceluiasi operator. Prin folosirea operatorilor supradefiniți, textul programului poate deveni foarte natural și ușor de înțeles. Acest capitol analizează în detaliu suprapunerea operatorilor și a funcțiilor. Când veți termina de citit capitolul, veți constata că suprapunerea operatorilor este un concept ușor și puternic, precum și că ați aflat următoarele:

- ♦ Ce reprezintă suprapunerea funcțiilor
- ♦ Când două funcții distincte au același nume, cum determină C++ alegerea funcției ce va fi apelată
- ♦ Cum se îmbunătățește claritatea programului prin suprapunerea funcțiilor



- Utilizarea parametrilor prestabiliți
- Ce-i de făcut când se folosesc parametri, alții decât cei prestabiliți
- Cum se creează o bibliotecă de funcții
- Ce reprezintă suprapunerea operatorilor
- Cum determină compilatorul C++ operația dorită
- Cum se supraîncarcă un operator
- Ce operatori permit suprapunerea în C++
- Cum se folosește un operator suprapus
- Cum se creează un operator suprapus cu acces la structurile membru ale unei clase

### SĂ ÎNȚELEGEM SUPRAPUNEREA FUNCȚIILOR

Înainte de a examina suprapunerea operatorilor, este important de a înțelege suprapunerea funcțiilor. Așa cum veți vedea, ambele concepte sunt destul de asemănătoare și implementate de compilatoarele C++. **Suprapunerea funcțiilor** este procesul de definire a două sau mai multor funcții folosind același nume, care diferă numai prin tipul de date returnat sau prin numărul și tipul parametrilor. Compilatorul va determina care funcție trebuie apelată pe baza modului în care aceasta este folosită. De exemplu, următorul program, OVERLOAD.CPP, creează două funcții denumite *show\_message*. Prima funcție nu are parametri, ci pur și simplu afișează un mesaj. A doua funcție primește ca parametru un șir de caractere:

```
#include <iostream.h>

void show_message(void)
{
    cout << "Success with C++!" << endl;
}

void show_message(char *message)
{
    cout << message;
}

void main(void)
{
    show_message("My favorite book is: ");
    show_message();
}
```

La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> OVERLOAD <ENTER>
My favorite book is: Success with C++!
```

După cum se poate vedea, primul apel de funcție face referință la acea definiție *show\_message* care suportă un șir de caractere drept parametru, în timp ce al doilea apel se referă la funcția care afișează mesajul prestabilit. La compilare și execuție, compilatorul C++ va determina care funcție trebuie apelată, pe baza numărului de parametri.



### SĂ ÎNȚELEGEM SUPRAPUNEREA FUNCȚIILOR

Suprapunerea funcțiilor este procesul prin care se definesc în program două sau mai multe funcții cu același nume. Funcțiile diferă numai prin numărul sau tipul parametrilor. La compilarea programului, C++ va determina univoc care din funcții trebuie apelată, pe baza parametrilor fiecărei funcții. În acest mod, programul va apela întotdeauna funcția corectă.

**Observație:** Deși suprapunerea funcțiilor poate fi un instrument foarte puternic și oportun, folosirea în exces a procedurii poate duce, totuși, la micșorarea clarității programului. Astfel, dacă definiți mai multe versiuni ale aceleiași funcții, estimați utilizarea suprapunerii funcțiilor pentru a vă asigura că programul respectiv este încă ușor de citit și înțeles.

### ANALIZA ALTOR EXEMPLE

Așa cum am văzut, compilatorul C++ determină ce funcție trebuie apelată, în funcție de parametrii transferați. Fie următorul program, INTARRAY.CPP, ce folosește funcția *sum\_values* pentru a calcula suma valorilor unui tablou de întregi:

```
#include <iostream.h>

long sum_array(int *array, int num_elements)
{
    long sum = 0L;

    for (int i = 0; i < num_elements; i++)
        sum += array[i];

    return(sum);
}

void main(void)
{
    ...
}
```

```
int array[5] = {1, 2, 3, 4, 5};

cout << "The values sum to " << sum_array(array, 5) << endl;
}
```

La compilarea și execuția programului, pe ecran va apărea:

```
C:\> INTARRAY <ENTER>
The values sum to 15
```

Să presupunem acum că dorim să însumăm valorile unui tablou de numere reale, definit în continuare:

```
float float_array[5] = {1.1, 2.2, 3.3, 4.4, 5.5};
```

Dacă se folosea limbajul C, era necesară crearea a două funcții cu nume diferite pentru însumarea valorilor din cele două tablouri, cum se arată în programul C\_ARRAYS.C de mai jos:

```
#include <stdio.h>

float sum_float_array(float *array, int num_elements)
{
    float sum = 0.0;
    int i;

    for (i = 0; i < num_elements; i++)
        sum += array[i];

    return(sum);
}

long sum_int_array(int *array, int num_elements)
{
    long sum = 0L;
    int i;

    for (i = 0; i < num_elements; i++)
        sum += array[i];

    return(sum);
}

void main(void)
{
    int int_array[5] = {1, 2, 3, 4, 5};
    float float_array[5] = {1.1, 2.2, 3.3, 4.4, 5.5};

    printf("Values in int array: %ld\n",
           sum_int_array(int_array, 5));
    printf("Values in float array: %f\n",
           sum_float_array(float_array, 5));
}
```

După cum se poate vedea, programul definește funcțiile *sum\_int\_array* și *sum\_float\_array* pentru a însuma valorile de tip *int* și *float* din cele două tablouri. Deși programul este corect, faptul că trebuie să creăm două funcții distincte pentru fiecare tip este destul de incomod și reduce claritatea programului. Folosind suprapunerea funcțiilor din C++, putem să folosim un singur nume de funcție, *sum\_array*, și să transferăm drept parametri ai acestei funcții tablouri de fiecare tip. Compilatorul va putea determina care funcție trebuie apelată. Următorul program, SUMARRAY.CPP, realizează acest lucru.

```
#include <iostream.h>

float sum_array(float *array, int num_elements)
{
    float sum = 0.0;

    for (int i = 0; i < num_elements; i++)
        sum += array[i];

    return(sum);
}

long sum_array(int *array, int num_elements)
{
    long sum = 0L;

    for (int i = 0; i < num_elements; i++)
        sum += array[i];

    return(sum);
}

void main(void)
{
    int int_array[5] = {1, 2, 3, 4, 5};
    float float_array[5] = {1.1, 2.2, 3.3, 4.4, 5.5};

    cout << "Values in int array: " << sum_array(int_array, 5) <<
        endl;
    cout << "Values in float array: " << sum_array(float_array, 5) <<
        endl;
}
```

După cum se observă, programul folosește un singur nume de funcție, *sum\_array*, pentru tablouri de tip *int* și *float*. Este important de reținut că trebuie totuși definite două funcții, câte una pentru fiecare tip, dar programul poate folosi același nume de funcție pentru fiecare tip de tablou.

În programul precedent, compilatorul C++ a putut determina, pe baza tipului de parametri, care din funcțiile *sum\_array* să fie apelată. În programul următor, GETLINE.CPP, compilatorul va determina care funcție *getline* să fie apelată, pe

baza numărului de parametri. În Capitolul 1 ați învățat că funcția membru *getline* permite streamului de intrare *cin* să citească o linie de text de la tastatură sau de la dispozitivul standard de intrare:

```
cin.getline(line, sizeof(line));
```

Când folosiți *getline*, puteți de asemenea, specifica un caracter la care operația de intrare să se încheie. De exemplu, următorul apel de funcție citește fie o linie de text, fie o parte a liniei de text, până la întâlnirea caracterului 'x' exclusiv:

```
cin.getline(line, sizeof(line), 'X');
```

Următorul program, GETLINE.CPP, folosește suprapunerea funcțiilor pentru a crea două funcții ce suportă aceste două operații:

```
#include <iostream.h>

void getline(char *line, int size)
{
    char letter;

    for (int i = 0; i < size; i++)
        if ((letter = cin.get()) == '\n')
            break;
        else
            line[i] = letter;

    line[i] = NULL;
}

void getline(char *line, int size, char terminator)
{
    char letter;

    for (int i = 0; i < size; i++)
        if ((letter = cin.get()) == '\n')
            break;
        else if (letter == terminator)
            break;
        else
            line[i] = letter;

    line[i] = NULL;
}

void main(void)
{
    char text[256];

    cout << "Type in a line of text: ";
    getline(text, sizeof(text));
    cout << "You typed: " << text << endl;
```

```
cout << "Type text followed by X: ";
getline(text, sizeof(text), 'X');
cout << "You typed: " << text << endl;
```

Așa cum se poate vedea, în program se definesc două funcții *getline* distincte. În timpul compilării, C++ determină corect funcția ce trebuie apelată. Funcția *getline* definită în acest program nu afectează funcția membru *cin.getline*, datorită regulilor de vizibilitate discutate în Capitolul 9.

În sfârșit, următorul program, SHOWDATE.CPP, creează câteva funcții denumite *show\_date*. Prima dintre ele afișează data sistem curentă. A doua afișează data specificată într-un șir de caractere, iar a treia afișează data conținută într-o structură:

```
#include <iostream.h>
#include <time.h>

void show_date(void)
{
    time_t current_datetime;

    time(&current_datetime);

    cout << "Current date: " << ctime(&current_datetime);
}

void show_date(char *date)
{
    cout << "String date: " << date << endl;
}

struct Date {
    int month;
    int day;
    int year;
};

void show_date(struct Date date)
{
    cout << "Structure date: ";

    switch (date.month) {
        case 1: cout << "January ";
                break;
        case 2: cout << "February ";
                break;
        case 3: cout << "March ";
                break;
```

```

case 4: cout << "April ";
        break;
case 5: cout << "May ";
        break;
case 6: cout << "June ";
        break;
case 7: cout << "July ";
        break;
case 8: cout << "August ";
        break;
case 9: cout << "September ";
        break;
case 10: cout << "October ";
        break;
case 11: cout << "November ";
        break;
case 12: cout << "December ";
        break;
};

cout << date.day << ", " << date.year << endl;
}

void main(void)
{
    struct Date date = {9, 30, 1994};
    char datestr[9];

    show_date(); // Show default

    show_date(_strdate(datestr));

    show_date(date);
}

```

După cum se observă, programul folosește 3 funcții diferite, toate având numele *show\_date*. În funcție de parametri pe care programul îi transferă la apelul funcțiilor, compilatorul C++ va determina care din funcții va fi apelată. La compilarea și execuția acestui program, pe ecran va apărea:

```

C:\> SHOWDATE <ENTER>
Current date: Wed Nov 24 19:30:03 1993
String date: 11/24/93
Structure date: September 30, 1994

```

**Observație:** Pentru simplitate, programul precedent a folosit o instrucțiune *case* pentru a afișa numele lunilor conținute în câmpul *date.month*. Printr-o schimbare simplă de program se poate folosi un tablou de pointeri la șiruri de caractere pentru afișarea lunilor, ca mai jos:

```

void show_date(struct Date date)
{
    char *months[] = { "January", "February", "March", "April",
                       "May", "June", "July", "August", "September",
                       "October", "November", "December" };

    cout << "Structure date: ";

    cout << months[date.month] << " " << date.day << ", " <<
        date.year << endl;
}

```

Așa cum se observă, prin definirea în acest mod a tabloului de șiruri de caractere, se reduce numărul de instrucțiuni din funcție, ceea ce duce la îmbunătățirea clarității codului.

### FOLOSIREA PARAMETRILOR PRESTABILIȚI

Pe lângă posibilitatea de a suprapune funcțiile, C++ permite, de asemenea, specificarea *valorilor parametrilor implicați* ai unei funcții. În acest fel, dacă programul apelează o anumită funcție și omite unul sau mai mulți parametri, funcția va folosi valori prestabilite. Pentru a înțelege mai bine cum funcționează mecanismul parametrilor prestabiliți, să considerăm programul *DEF\_PAR.CPP* care definește funcția *show\_values*. Funcția are 3 parametri, iar dacă este apelată fără ca toți parametrii să fie specificați, se vor folosi valorile prestabilite acolo unde este necesar:

```

#include <iostream.h>

void show_values(int a = 1, int b = 2, int c = 3)
{
    cout << a << " " << b << " " << c << endl;
}

void main(void)
{
    show_values();
    show_values(1001);
    show_values(1001, 2002);
    show_values(1001, 2002, 3003);
}

```

După cum se vede, programul apelează funcția de patru ori, folosind patru combinații diferite de parametri. La compilarea și execuția programului, pe ecran se va afișa:

```

C:\> DEF_PAR <ENTER>
1 2 3
1001 2 3

```

1001 2002 3  
1001 2002 3003

Se observă că, atunci când programul nu specifică valori pentru parametri, funcția folosește parametrii prestabiliți.

Când se omite un parametru al unei funcții care acceptă parametri prestabiliți, trebuie omise valorile tuturor celorlalți parametri care urmează. De exemplu, în programul precedent, nu se putea preciza valoarea parametrului *a*, pentru a se omite valoarea pentru parametrul *b*, iar apoi să se specifice valoarea pentru parametrul *c*, ca mai jos:

```
show_values(1001, , 3003);
```

Eroare de sintaxă la omiterea parametrului din mijloc

În cazul precedent, dacă programul omite valoarea pentru parametrul *b*, el trebuie să omită și valoarea pentru parametrul *c*.

Anterior ați creat, în acest capitol, două versiuni ale funcției *getline*. După cum probabil vă amintiți, prima funcție citea o linie de text de la tastatură până la întâlnirea caracterului CR exclusiv. A doua funcție permitea programului să specifice un caracter de încheiere. Următorul program, DEF\_LINE.CPP, modifică programul anterior pentru a folosi o valoare prestabilită de parametru pentru caracterul de încheiere. În acest mod, programul poate atinge același scop folosind o singură funcție:

```
#include <iostream.h>
```

```
void getline(char *line, int size, char terminator = '\n')
{
    char letter;

    for (int i = 0; i < size; i++)
        if ((letter = cin.get()) == terminator)
            break;
        else
            line[i] = letter;

    line[i] = NULL;
}
```

```
void main(void)
{
    char text[256];

    cout << "Type in a line of text: ";
    getline(text, sizeof(text));
    cout << "You typed: " << text << endl;
```

```
cout << "Type text followed by X: ";
getline(text, sizeof(text), 'X');
cout << "You typed: " << text << endl;
```

Așa cum se observă, programul folosește acum o singură funcție *getline*. Dacă programul va apela funcția *getline* fără a se specifica caracterul de încheiere, va fi folosit, în mod prestabilit, caracterul '\n'. În cazul specificării caracterului de încheiere drept parametru, programul va citi până când se va umple bufferul sau până la întâlnirea caracterului respectiv. În acest mod, programul va putea citi caractere și după întâlnirea caracterului '\n'. Astfel se poate folosi funcția *getline* pentru a citi de la tastatură caractere care includ și caracterul '\n'.

Este interesant de remarcat că, dacă examinați definiția clasei *istream* în fișierul antet folosit de compilatorul C++ pentru funcții stream, veți găsi pentru funcția *getline* un prototip de forma:

```
istream & _FAR & _Cdecl getline(unsigned char _FAR *, int, char = '\n');
```

Se observă că funcția specifică caracterul sfârșit de linie drept valoare prestabilită pentru parametrul ce indică încheierea citirii.



### SĂ ÎNȚELEM PARAMETRII PRESTABIȚI

Când scrieți funcții în C++, este posibil ca majoritatea programelor să folosească aceleași valori pentru anumiți parametri. Pentru simplificarea folosirii acestor funcții, se pot specifica valori prestabilite pentru parametri. O valoare prestabilită este specificată în antetul funcției, fiind precedată de semnul egal. De exemplu, următoarea funcție *getline* specifică caracterul *newline* drept parametru prestabilit:

```
void getline (char * text, int size, char terminator = '\n')
```

Dacă programul apelează funcția *getline* numai cu doi parametri, al treilea parametru va avea valoarea prestabilită:

```
getline (line, sizeof (line));
```

Dacă programul specifică valoarea de terminare, valoarea implicită va fi neglijată:

```
getline (line, sizeof (line), 'x');
```

O funcție poate avea mai mulți parametri implicați. De exemplu, funcția *get\_tax* folosește doi parametri prestabiliți:

```
float get_tax (float amount, float sales_tax = 0.06,
              float state_tax = 0.03);
```

În acest caz, programul poate omite valorile pentru parametrii *sales\_tax* și *state\_tax*. Când programul omite valoarea pentru un parametru, el trebuie să omită valorile pentru toți parametrii care urmează. În cazul de mai sus, dacă programul omite valoarea pentru parametrul *sales\_tax*, el trebuie să omită valoarea și pentru parametrul *state\_tax*.

### CUM SĂ CREĂM PROPRIILE BIBLIOTECI DE FUNCȚII

Pe măsură ce numărul funcțiilor crește, vom dori să creăm propriile noastre *biblioteci de funcții*, în care programele să caute funcțiile de care au nevoie. Pentru a facilita folosirea funcțiilor, trebuie creat un fișier antet, care să conțină prototipul fiecărei funcții. La funcțiile care suportă parametri prestabiliți, valorile corespunzătoare trebuie specificate în fișierul antet.

### SĂ ÎNȚELEM SUPRAPUNEREA OPERATORILOR

*Suprapunerea operatorilor* este procesul de atribuire a două sau mai multor operații aceluiași operator. În funcție de modul de utilizare a operatorului, compilatorul C++ va determina operația care se va realiza. Pentru a înțelege mai bine acest concept, să examinăm un program care creează o clasă *stack* cu care se pot depune valori în memorie (*push*) sau se pot scoate (*pop*) din memorie. Clasa *stack* (stivă) are următoarea formă:

```
class stack {
public:
    stack(int size);
    int push(int value);
    int pop(void);
    int is_empty(void) { return(isempty); };
    int is_full(void) { return(isfull); };
private:
    int *storage; // Stack data buffer
    int elements; // Number of values in the stack
    int isempty; // True when stack is empty
    int isfull; // True when stack is full
    int stack_size; // Number of values the stack can store
};
```

Următorul program, *STACK.CPP*, creează un obiect de tipul *stack* ce poate conține 64 de valori. Programul depune apoi valori în stivă, până când aceasta se umple. Ulterior, programul scoate valorile din stivă, una câte una:

```
#include <iostream.h>
```

```
class stack {
```

```
public:
    stack(int size);
    int push(int value);
    int pop(void);
    int is_empty(void) { return(isempty); };
    int is_full(void) { return(isfull); };
private:
    int *storage; // Stack data buffer
    int elements; // Number of values in the stack
    int isempty; // True when stack is empty
    int isfull; // True when stack is full
    int stack_size; // Number of values the stack can store
};

stack::stack(int size)
{
    storage = new int[size]; // Allocate memory for the stack
    elements = 0;
    isempty = 1;
    isfull = 0;
    stack_size = size;
}

int stack::pop(void)
{
    if (is_empty())
        return(0);
    else
    {
        if (--elements == 0)
            isempty = 1;

        isfull = 0;

        return(storage[elements]);
    }
}

int stack::push(int value)
{
    if (is_full())
        return(0);
    else
    {
        isempty = 0;
        storage[elements++] = value;

        if (elements == stack_size)
            isfull = 1;

        return(value);
    }
}
```

```

    }
}

void main(void)
{
    stack fifo(64);

    int i;

    for (i = 0; ! fifo.is_full(); i++)
        fifo.push(i);

    while (! fifo.is_empty())
        cout << fifo.pop() << endl;
}

```

După cum se poate vedea, programul folosește funcțiile membru *push* și *pop* pentru a depune, respectiv pentru a scoate elemente din stivă. Pentru a facilita citirea și înțelegerea programelor, se pot supradefini operatorii + și -- pentru clasa *stack*. De exemplu, operatorul + s-ar putea folosi în locul funcției *push*, iar operatorul -- în locul funcției *pop*:

```

pushed_value = fifo + 54;
fifo = --fifo;

```

La suprapunerea unui operator, se specifică funcția pe care programul o execută când compilatorul C++ întâlnește operatorul respectiv. De exemplu, când C++ întâlnește operatorul + cu obiectele de tipul *stack*, compilatorul va insera cod pentru apelarea funcției și nu va realiza o adunare efectivă. Pentru a defini o funcție operator, se definește o funcție obișnuită, exceptând includerea cuvântului-cheie *operator*. Următoarea funcție, de exemplu, suprapune operatorul +:

Tip valoare returnată      Nume clasă

```

int stack::operator +(int value)
{
    if (is_full())
        return(0);
    else
    {
        isempty = 0;
        storage[elements++] = value;

        if (elements == stack_size)
            isfull = 1;

        return(value);
    }
}

```

Parametru  
Operator  
Cuvânt cheie operator

Analog, următoarea funcție supradefinește operatorul --:

```

int stack::operator -(void)
{
    if (is_empty())
        return(0);
    else
    {
        if (--elements == 0)
            isempty = 1;

        isfull = 0;

        return(storage[elements]);
    }
}

```

La definirea operatorilor unei clase, aceștia trebuie incluși în definiția clasei, după cum se arată mai jos:

```

class stack {
public:
    stack(int size);
    int operator +(int value);
    int operator -(void);
    int is_empty(void) { return(isempty); };
    int is_full(void) { return(isfull); };
private:
    int *storage; // Stack data buffer
    int elements; // Number of values in the stack
    int isempty; // True when stack is empty
    int isfull; // True when stack is full
    int stack_size; // Number of values the stack can store
};

```

Operatori clasă  
de tip public

Următorul program, STACKOVR.CPP folosește operatorii + și -- pentru a plasa și a scoate valori din stivă:

```

#include <iostream.h>

class stack {
public:
    stack(int size);
    int operator +(int value);
    int operator -(void);
    int is_empty(void) { return(isempty); };
    int is_full(void) { return(isfull); };
private:
    int *storage; // Stack data buffer
    int elements; // Number of values in the stack
};

```



```

int isempty;           // True when stack is empty
int isfull;            // True when stack is full
int stack_size;        // Number of values the stack can store
};

stack::stack(int size)
{
    storage = new int[size]; // Allocate memory for the stack
    elements = 0;
    isempty = 1;
    isfull = 0;
    stack_size = size;
}

int stack::operator--(void)
{
    if (isempty())
        return(0);
    else
    {
        if (--elements == 0)
            isempty = 1;

        isfull = 0;

        return(storage[elements]);
    }
}

int stack::operator+(int value)
{
    if (is_full())
        return(0);
    else
    {
        isempty = 0;
        storage[elements++] = value;

        if (elements == stack_size)
            isfull = 1;

        return(value);
    }
}

void main(void)
{
    stack fifo(64);

    int i;

```

```

for (i = 0; ! fifo.is_full(); i++)
    fifo + i;

while (! fifo.is_empty())
    cout << -fifo << endl;
}

```

Așa cum se observă, programul folosește următoarea instrucțiune pentru a depune valori în stivă:

```
fifo + i;
```

Pentru afișarea valorii unui element scos din stivă, se folosește instrucțiunea:

```
cout << -fifo << endl;
```



### SĂ ÎNȚELEM SUPRAPUNEREA OPERATORILOR

Suprapunerea operatorilor este procesul de atribuire a două sau mai multor operații aceluiași operator. Prin suprapunerea unor operatori, programele pot exprima operațiile din interiorul claselor într-un mod mai natural. De exemplu, operatorul + poate fi folosit pentru adunare obișnuită, pentru a insera zile într-o dată de tip structură, sau pentru a concatena două șiruri de caractere, cum se arată mai jos:

```
value = some_value + 100;
```

```
expiration_date = sales_date + 30; // date class
```

```
new_string = "Success " + "with C++!"; // string class
```

Pentru a suprapune un operator, se definește o funcție pe care compilatorul C++ o apelează ori de câte ori găsește acele tipuri de date pentru care a fost definit operatorul. Definiția funcției operator este asemănătoare unei funcții obișnuite, cu excepția faptului că trebuie inclus cuvântul cheie *operator* în antetul funcției.

### REGULI PENTRU SUPRAPUNEREA OPERATORILOR

La suprapunerea unui operator, se specifică de fapt două tipuri de date pentru care se dorește o tratare diferită a operației respective. Astfel, un operator poate fi folosit numai în modul în care programul îl utilizează în mod obișnuit. De exemplu, nu se poate folosi un operator unar, cum este -- (operator de decrementare), pentru a realiza o operație pe două valori:

```
result = a - b; // Syntax error
```

În general, C++ permite suprapunerea tuturor operatorilor, cu excepția celor prezentați în Tabela 5.1:

Operator	Scop	Exemplu
.	membru al unei clase sau al unei structuri	cin.get()
.*	pointer la membru	object.*member
::	operator de rezoluție globală	class_name::member
?:	operatorul condițional	c = (a > b)?a:b

Tabela 5.1. Operatorii ce nu pot fi supradefiniți în C++.

## SUPRAPUNEREA OPERATORILOR I/O

În programul precedent am suprapus operatorii + și - – pentru a-i folosi cu obiecte de tip stivă. În funcție de modul de utilizare a claselor, pot apare și situații când se dorește suprapunerea operatorilor de inserție (<<) și extracție (>>). Să considerăm următorul program, BOOKONE.CPP, care folosește clasa *book* pentru a memora informații despre o anumită carte:

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>

class book {
public:
    book(char *title, char *author, char *publisher, float price);
    void show_book(void);
private:
    char title[64];
    char author[64];
    char publisher[64];
    float price;
};

book::book(char *title, char *author, char *publisher, float price)
{
    strcpy(book::title, title);
    strcpy(book::author, author);
    strcpy(book::publisher, publisher);
    book::price = price;
}

void book::show_book(void)
{
```

```
    cout << "Title: " << title << endl;
    cout << "Author: " << author << endl;
    cout << "Publisher: " << publisher << endl;
    cout << setprecision(2) << "Price: " << price << endl;
}

void main(void)
{
    book computer_book("Success with C++", "Kris Jamsa",
        "Jamsa Press", 29.95);

    computer_book.show_book();
}
```

După cum se observă, programul folosește funcția membru *show\_book* pentru a afișa informația despre carte. În cadrul funcției, se folosește operatorul de inserție cu *cout*. Programul ar putea elimina funcția membru *show\_book* prin suprapunerea operatorului de extracție (<<) pentru a-l utiliza la obiecte de tipul *book*. În acest mod, programul va putea afișa informația despre carte folosind operatorul de extracție astfel:

```
cout << computer_book;
```

Pentru a suprapune un operator, programul trebuie să creeze o funcție care se execută de fiecare dată când operatorul este folosit cu clasa respectivă. Următoarele instrucțiuni suprapun operatorul de extracție pentru a fi folosit cu obiecte de tipul *book*:

```
ostream& operator <<(ostream&stream, book bookinfo)
{
    cout << "Title: " << bookinfo.title << endl;
    cout << "Author: " << bookinfo.author << endl;
    cout << "Publisher: " << bookinfo.publisher << endl;
    cout << setprecision(2) << "Price: " << bookinfo.price << endl;
    return(stream);
}
```

După cum se vede, definiția operatorului este în mare parte similară funcției *show\_book* folosită anterior, prin aceea că diferite câmpuri ale obiectului *book* sunt scrise în *cout*.

Cea mai confuză parte a definiției este, de departe, antetul funcției:

```
ostream& operator <<(ostream&stream, book bookinfo)
```

Pentru a înțelege această instrucțiune, să ne reamintim că, la suprapunerea unui operator se definește o funcție pe care programul o execută când este întâlnit operatorul. În acest caz, funcția returnează o referință la un obiect de tipul *ostream*. Parametrii operației sunt un obiect *ostream* (probabil *cout*) și un obiect de tipul *book*:

Tip valoare returnată

```
ostream& operator <<(ostream& stream, book bookinfo)
```

Parametri funcție

Pentru a înțelege parametrii funcției, să ne gândim la operanzii care apar în partea stângă și în partea dreaptă a operatorului:

Obiect ostream

```
cout << computer_book;
```

Obiect book

În sfârșit, după ce funcția inserează informațiile în stream-urile de ieșire, se folosește instrucțiunea *return* pentru a returna stream-ul actualizat:

```
return(stream);
```

Dacă nu înțelegi prea bine referințele la stream-urile I/O, nu vă faceți griji. În Capitolul 10 vom discuta referințele C++ în detaliu.

Următorul program, BOOKTWO.CPP, folosește suprapunerea operatorilor pentru a afișa informația despre respectivul obiect de tip *book*:

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>

class book {
public:
    book(char *title, char *author, char *publisher, float price);
    friend ostream& operator <<(ostream& stream, book bookinfo);
private:
    char title[64];
    char author[64];
    char publisher[64];
    float price;
};

book::book(char *title, char *author, char *publisher, float price)
{
    strcpy(book::title, title);
    strcpy(book::author, author);
    strcpy(book::publisher, publisher);
    book::price = price;
}

ostream& operator <<(ostream& stream, book bookinfo)
{
    cout << "Title: " << bookinfo.title << endl;
    cout << "Author: " << bookinfo.author << endl;
    cout << "Publisher: " << bookinfo.publisher << endl;
```

```
cout << setprecision(2) << "Price: " << bookinfo.price << endl;
return(stream);
}
```

```
void main(void)
{
    book computer_book("Success with C++", "Kris Jamsa",
        "Jamsa Press", 29.95);

    cout << "Book Information" << endl;
    cout << computer_book;
}
```

După cum se observă, programul afișează informații despre obiectul *book* folosind operatorul de inserție. De notat că programul folosește, de asemenea, operatorul de inserție pentru a scrie un mesaj la *cout* imediat înainte de a afișa variabilele membru ale obiectului.

```
cout << "Book Information" << endl;
cout << computer_book;
```

Cele două linii s-ar fi putut chiar combina sub forma următoare:

```
cout << "Book Information" << endl << computer_book;
```

Rețineți că operatorul suprapus este folosit numai cu obiecte de tip *book*. Astfel, prima instrucțiune folosește operatorul de inserție standard, în timp ce a doua instrucțiune folosește același operator suprapus. În acest fel, un operator suprapus este similar unei funcții suprapuse. Să mai observăm că, în cadrul definiției clasei, operatorul este suprapus ca o funcție *friend*:

```
class book {
public:
    book(char *title, char *author, char *publisher, float price);
    friend ostream& operator <<(ostream& stream, book bookinfo);
private:
    char title[64];
    char author[64];
    char publisher[64];
    float price;
};
```

Operator declarat ca friend

Declarând operatorul de inserție ca o funcție *friend*, clasa îi acordă dreptul de acces la membrii săi de tip *private*.

## ANALIZA ALTOR EXEMPLE

Cu cât programele folosesc mai mult suprapunerea operatorilor, cu atât mai mult veți aprecia puterea și facilitățile acestui concept. De aceea, în această secțiune vom mai examina câteva programe ce supradefinesc operatorii pentru diferite

tipuri de clase. Experimentați cât mai mult aceste programe. Veți vedea că suprapunerea operatorilor nu este dificilă, o dată ce ați practicat-o de câteva ori.

Următorul program, DATE\_OPS.CPP, suprapune operatorii de inserție și extracție astfel încât să se poată realiza operații de intrare/ieșire pe structura *Date*:

```
#include <iostream.h>
#include <string.h>

class Date {
public:
    Date(int month, int day, int year);
    friend ostream& operator <<(ostream& stream, Date date);
    friend istream& operator >>(istream& stream, Date *date);
private:
    int month;
    int day;
    int year;
};

Date::Date(int month, int day, int year)
{
    Date::month = month;
    Date::day = day;
    Date::year = year;
}

ostream& operator <<(ostream& stream, Date date)
{
    stream << date.month << '/' << date.day << '/' << date.year << endl;
    return(stream);
}

istream& operator >>(istream& stream, Date *date)
{
    stream >> date->month;
    stream >> date->day;
    stream >> date->year;
    return(stream);
}

void main(void)
{
    Date christmas(12, 25, 94);
    Date birthday(9, 30, 94);

    cout << "Christmas is " << christmas;
```

```
    cout << "Type in your birthday mm dd yy: ";
    cin >> &birthday;
    cout << "You typed: " << birthday;
}
```

Se observă că, în cadrul clasei *Date*, operatorii de inserție și extracție sunt declarați drept *friend*, pentru a avea acces la variabilele membru de tip private:

```
class Date {
public:
    Date(int month, int day, int year);
    friend ostream& operator <<(ostream& stream, Date date);
    friend istream& operator >>(istream& stream, Date *date);
private:
    int month;
    int day;
    int year;
};
```

Declararea funcțiilor operator ca friend

Funcțiile operator suprapuse returnează o referință la un stream, primind drept parametri stream-ul și clasa:

```
ostream& operator <<(ostream& stream, Date date)
{
    stream << date.month << '/' << date.day << '/' <<
        date.year << endl;
    return(stream);
}

istream& operator >>(istream& stream, Date *date)
{
    stream >> date->month;
    stream >> date->day;
    stream >> date->year;

    return(stream);
}
```

Returnează o referință la stream  
Parametru stream

Structură date

Returnează o referință la stream

Parametru stream

Pointer la structură de date

Deoarece operatorul de extracție suprapus atribuie valori variabilelor clasei, funcția trebuie să lucreze cu un pointer la clasă. În cazurile când în programe se lucrează cu clase sau structuri ce conțin mai multe valori, cum ar fi o dată, apar situații când este nevoie de a testa egalitatea a două clase sau două structuri. Următorul program TEST\_OPS.CPP, suprapune operatorul de egalitate (==) și inegalitate (!=) pentru lucrul cu clase de tipul *Date*. Folosind acești operatori, programul poate determina rapid dacă două clase conțin sau nu aceleași valori.

```

#include <iostream.h>
#include <string.h>

class Date {
public:
    Date(int month, int day, int year);
    friend int operator ==(Date date_one, Date date_two);
    friend int operator !=(Date date_one, Date date_two);
private:
    int month;
    int day;
    int year;
};

Date::Date(int month, int day, int year)
{
    Date::month = month;
    Date::day = day;
    Date::year = year;
}

int operator ==(Date date_one, Date date_two)
{
    if (date_one.day != date_two.day)
        return(0);
    else if (date_one.month != date_two.month)
        return(0);
    else if (date_one.year != date_two.year)
        return(0);
    return(1);
}

int operator !=(Date date_one, Date date_two)
{
    if (date_one.day != date_two.day)
        return(1);
    else if (date_one.month != date_two.month)
        return(1);
    else if (date_one.year != date_two.year)
        return(1);
    return(0);
}

void main(void)
{
    Date christmas(12, 25, 94);
    Date santas_day(12, 25, 94);
    Date birthday(9, 30, 94);

    if (christmas == santas_day)
        cout << "Christmas and Santa's day are the same" << endl;
}

```

```

if (christmas != birthday)
    cout << "My birthday is not Christmas" << endl;
}

```

Prin suprapunerea operatorilor == și !=, programul devine mult mai sugestiv.

Alternativa testării egalității a două clase de tipul *Date* ar fi putut fi:

```

if ((christmas.day == birthday.day) &&
    (christmas.month == birthday.month) &&
    (christmas.year == birthday.year))
    cout << "My birthday is on Christmas!" << endl;

```

În acest caz, nu numai că secvența de cod devine mai dificil de înțeles, dar instrucțiunile suplimentare măresc posibilitatea apariției unei erori în program. Examinând programele din această carte, veți observa cum suprapunerea operatorilor îmbunătățește claritatea programelor și capacitatea de înțelegere a acestora de către programator.

## REZUMAT

Suprapunerea operatorilor este procesul atribuirii a două sau mai multor operații aceleiași funcții. În acest capitol ambele concepte s-au examinat în detaliu. Înainte de a trece la Capitolul 6, asigurați-vă că ați învățat următoarele:

- ✓ Suprapunerea funcțiilor este procesul de definire a două sau mai multor funcții cu același nume, dar care diferă prin tipul rezultatului sau al parametrilor.
- ✓ Compilatorul C++ va determina apelarea unei anumite funcții pe baza tipului rezultatului sau al parametrilor.
- ✓ Suprapunerea funcțiilor permite îmbunătățirea clarității programului, prin folosirea unor funcții cu același nume pentru aceleași operații, funcții care diferă numai prin tipul de date folosit.
- ✓ Pentru a mări flexibilitatea folosirii funcțiilor, compilatorul C++ suportă parametri prestabiliți. Dacă la apelul funcțiilor nu se specifică toți parametrii actuali, atunci C++ îi va substitui cu valorile prestabilite.
- ✓ Dacă la apelul unei funcții se omite un parametru, toți parametrii care urmează trebuie omiși. Cu alte cuvinte, nu se poate specifica o valoare pentru parametrul 1, urmată de valoarea prestabilită a parametrului 2 și din nou de o valoare a parametrului 3.
- ✓ Pentru a crea o bibliotecă de funcții, se creează un fișier antet ce conține prototipurile fiecărei funcții. Dacă funcțiile suportă parametri prestabiliți, aceștia se specifică în fișierul prototip.
- ✓ Suprapunerea operatorilor este atribuirea a două sau mai multor operații aceluiași operator. În timpul compilării, C++ deter-

mină operația ce trebuie efectuată, pe baza modului de utilizare a operanzilor.

- ✓ Pentru a suprapune un operator, programul specifică o funcție pe care compilatorul o apelează când întâlnește operatorul.
- ✓ Compilatorul C++ permite suprapunerea majorității operatorilor, excepție făcând:

Operator	Nume
.	membru al clasei
*	pointer la membru la clasei
::	rezoluție globală
?:	atribuire condițională

- ✓ Dacă un operator este suprapus, el trebuie folosit într-o manieră consecvență față de modul normal de utilizare a acestuia.
- ✓ La suprapunerea unui operator pentru o clasă, de obicei acesta se declară ca o funcție *friend*, pentru a-i asigura accesul la membrii acelei clase.

## CAPITOLUL 6

### ACOMODAREA CU ȘABLOANELE

În Capitolul 5 ați învățat cum suprapunerea funcțiilor și operatorilor poate facilita înțelegerea programelor, prin folosirea unor nume de funcții sugestive. De exemplu, să presupunem că un program are de calculat valoarea medie a unor tablouri de tip *int* sau *float*. Prin suprapunerea numelui de funcție *average\_value*, programul poate folosi două funcții cu același nume, una pentru valorile întregi și alta pentru valorile reale. Dacă alți programatori vor citi programul, vor înțelege mai ușor scopul funcției *average\_value*, decât dacă ar fi existat două funcții cu nume diferite, ca *f\_average\_value* și *i\_average\_value* (cum ar fi fost necesar în C). Din păcate, cu toate că facilitatea de a folosi același nume de funcție este foarte convenabilă, codul trebuie scris pentru fiecare funcție în parte. În majoritatea cazurilor, singura diferență între astfel de funcții este tipul datelor cu care operează funcțiile.

Acest capitol tratează funcțiile-șablon în C++, care permit definirea unui șablon (template) sau a unei schițe ("blueprint") pentru definițiile de funcții. Folosind aceste șabloane, programele pot direcționa compilatorul C++ pentru a crea în mod automat codul unei funcții pentru diferite tipuri de variabile. De exemplu, un program ar putea crea un șablon *average\_value*. Folosind acest șablon, programul va putea crea funcții pentru tablouri de tip *int*, *float* și altele. Pe lângă șabloane de funcții, acest capitol tratează și șabloane de clase, ceea ce vă permite să definiți clase ale căror membri diferă numai prin tip.

La sfârșitul acestui capitol, veți ști următoarele:

- ♦ Ce este un șablon și la ce folosește
- ♦ C++ suportă șabloanele de funcții și clase
- ♦ Cum se creează un șablon de funcție
- ♦ Cum se creează ulterior o funcție ce returnează sau admite parametri de un anumit tip
- ♦ Cum atribuie compilatorul tipurile, dacă șablonul de funcție admite mai multe tipuri de parametri
- ♦ Cum se creează un șablon de clasă
- ♦ Cum se creează obiecte folosind un șablon de clasă

- Cum atribuie compilatorul tipurile, dacă șablonul de clasă admite mai multe tipuri de parametri
- Cum realizează compilatorul substituția de tipuri, dacă un șablon de clasă folosește alt șablon

## CREAREA PRIMULUI ȘABLON DE FUNCȚIE

Un *șablon de funcție* este un tipar din care compilatorul C++ construiește funcții pentru diferite tipuri de variabile. De exemplu, următoarele instrucțiuni creează un șablon pentru funcția *average\_value*.

```
template<class T> T average_value(T *array, int num_elements)
{
    T sum = 0;

    for (int i = 0; i < num_elements; i++)
        sum += array[i];

    return (sum / num_elements);
}
```

Înainte de a examina codul funcției șablon, priviți cu atenție la prima linie:

```
template<class T> T average_value(T *array, int num_elements)
```

Cuvântul cheie *template* informează compilatorul C++ că instrucțiunile care urmează definesc un șablon. Așa cum ați citit, șabloanele de funcții permit programelor să specifice instrucțiuni pentru funcții care diferă numai prin tip. Simbolul *<class T>* care urmează după cuvântul-cheie *template* specifică un simbol care reprezintă tipul funcției sau tipul parametrilor funcției. În acest exemplu, simbolul tipului este *T*. Restul liniei indică rezultatul funcției și tipul parametrilor, ca la o declarație de funcție obișnuită.

Să presupunem, de exemplu, că fiecare apariție a lui *T* va fi substituită prin tipul *float*. Antetul de funcție va fi atunci:

```
float average_value(float *array, int num_elements)
```

Analog, dacă *T* va fi substituit cu *int*, antetul de funcție devine:

```
int average_value(int *array, int num_elements)
```

Observați în corpul funcției instrucțiunea care folosește simbolul *T* pentru a declara variabila *sum*:

```
T sum = 0;
```

Când compilatorul C++ substituie pe *T* cu numele tipului în antetul de funcție, va substitui și simbolul specificat în corpul funcției. Pentru ca C++ să cunoască

tipul funcțiilor ce urmează a fi create, se pot specifica prototipuri de funcții imediat după definiția șablonului, ca mai jos:

```
int average_value(int *, int);
```

```
float average_value(float *, int);
```

În acest caz, compilatorul C++ va crea automat funcții pentru tipul *int* și *float*. După cum se vede în continuare, C++ folosește tipul primului parametru pentru a substitui pe *T*:

```
int average_value(int *, int);
```

```
template<class T> T average_value(T *array, int num_elements)
{
    T sum = 0;
```

```
    for (int i = 0; i < num_elements; i++)
        sum += array[i];
```

```
    return (sum / num_elements);
}
```

```
int average_value(int *array, int num_elements)
```

```
{
    int sum = 0;
```

```
    for (int i = 0; i < num_elements; i++)
        sum += array[i];
```

```
    return (sum / num_elements);
}
```

Pe lângă folosirea prototipurilor, programul poate folosi direct funcția, transferând acesteia parametrii ai căror tipuri determină tipurile corespunzătoare de funcții. Următorul program, *TEMP\_AVG.CPP* folosește șablonul *average\_value*:

```
#include <iostream.h>
```

```
template<class T> T average_value(T *array, int num_elements)
```

```
{
    T sum = 0;
```

```
    for (int i = 0; i < num_elements; i++)
        sum += array[i];
```

```
    return (sum / num_elements);
}
```

```
int average_value(int *, int);
```



```
float average_value(float *, int);

void main(void)
{
    int values[] = { 1, 2, 3, 4, 5 };
    float prices[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };

    cout << "Average of integer values is " <<
        average_value(values, 5) << endl;
    cout << "Average of floating-point values is " <<
        average_value(prices, 5) << endl;
}
```

La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> TEMP_AVG <ENTER>
Average of integer values is 3
Average of floating-point values is 3.3
```

Experimentați acest program cu răbdare, îndepărtând mai întâi prototipurile de funcții. La o nouă compilare a programului, C++ va determina corect tipurile pentru șablon, în funcție de parametri transferați funcției. În al doilea rând, se poate adăuga la program următoarea declarație de tablou:

```
long distances[] = { 1000000L, 2000000L, 3000000L };
```

Pentru a determina valoarea medie a tabloului, folosiți instrucțiunea:

```
cout << "Average of long values is " <<
    average_value(distances, 3) << endl;
```

**Observație:** Funcția șablon `average_value` concretizată pentru tipul `int` funcționează în programul precedent deoarece suma valorilor nu depășește suma variabilelor de tip `int`. Dacă se folosește, însă, următorul tablou, funcția va furniza un rezultat eronat:

```
int values[] = {1000, 5000, 10000, 15000, 20000};
```

Pentru a rezolva eroarea de depășire, tipul variabilei `sum` trebuie să fie `long`:

```
long sum = 0;
```

Din păcate, această declarație poate crea probleme pentru alte tipuri de date, cum ar fi un tablou de tip `float`. Tot în acest capitol veți învăța cum se specifică mai multe tipuri de parametri în funcțiile șablon, pentru situații ca acestea.

## SĂ ÎNȚELEM ȘABLOANELE DE FUNCȚII



Un șablon de funcție este un tipar după care compilatorul poate construi automat diverse funcții. Șabloanele de funcții sunt ideale pentru funcțiile care diferă numai prin tip. Pentru crearea unui șablon de funcție, se folosește cuvântul-cheie *template*, ca mai jos:

```
template<class T> T max_value(T *array, int num_elements)
{
    T max = array[0];

    for (int i = 0; i < num_elements; i++)
        if (max < array[i])
            max = array[i];

    return(max);
}
```

În acest exemplu, se creează un șablon pentru funcții cu numele `max_value`. Construcția `<class T>` specifică un simbol (aici este `T`) pe care compilatorul îl va substitui cu un tip concret de date. Pentru a indica compilatorului tipul dorit, se pot specifica în program prototipuri de funcții similare celor de mai jos:

```
float max_value(float *, int);
```

```
int max_value(int *, int);
```

În timpul compilării, simbolul `T` va fi înlocuit cu primul tip de parametri creându-se funcții suprapuse pentru tipul respectiv. Șabloanele de funcții au avantajul unei determinări rapide a modurilor în care este folosit șablonul. Dacă programul nu specifică prototipuri de funcții pentru un anumit șablon, compilatorul va putea să determine ce funcții să creeze, pornind de la modul de folosire a funcției în program.

Următorul program, `MIN_MAX.CPP`, creează și folosește șabloane pentru funcțiile `max_value` și `min_value`.

```
#include <iostream.h>
```

```
template<class T> T max_value(T *array, int num_elements)
{
    T max = array[0];

    for (int i = 0; i < num_elements; i++)
```

```

    if (max < array[i])
        max = array[i];

    return(max);
}

template<class T> T min_value(T *array, int num_elements)
{
    T min = array[0];

    for (int i = 0; i < num_elements; i++)
        if (min > array[i])
            min = array[i];

    return(min);
}

float max_value(float *, int);

int max_value(int *, int);

float min_value(float *, int);

int min_value(int *, int);

void main(void)
{
    int values[] = {5, 1, 6, 12, 7};

    float prices[] = { 1.1, 3.3, 4.4, 2.2, 5.5 };

    cout << "Max of integer values is " << max_value(values, 5) << endl;
    cout << "Max of floating-point values is " << max_value(prices, 5) << endl;

    cout << "Min of integer values is " << min_value(values, 5) << endl;
    cout << "Min of floating-point values is " << min_value(prices, 5) << endl;
}

```

La compilarea și execuția acestui program, pe ecran va apărea:

```

C:\> MIN_MAX <ENTER>
Max of integer values is 12
Max of floating-point values is 5.5
Min of integer values is 1
Min of floating-point values is 1.1

```

### CLARIFICAREA DEFINIȚIEI UNUI ȘABLON



Așa cum am văzut, cuvântul-cheie *template* permite definierea unor tipare pentru definiții de funcții. Când întâlnim pentru prima dată șabloane, declarația acestora poate fi puțin derutantă. De exemplu, următoarea instrucțiune începe cu o definiție de șablon pentru funcția *max\_value*:

```
template<class T> T max_value(T *array, int num_elements)
```

Pentru o mai bună claritate, se pot folosi două linii pentru a prezenta antetul șablonului, după cum urmează:

```

template<class T>
T max_value(T *array, int num_elements)
{
    T min = array[0];

    for (int i = 0; i < num_elements; i++)
        if (min > array[i])
            min = array[i];

    return(min);
}

```

După cum se observă, o dată ce se trece de primul rând al definiției șablonului, aceasta seamănă cu o definiție de funcție standard.

### ȘABLOANE CARE FOLOSESC MAI MULTE TIPURI

Ați învățat până acum că, atunci când C++ întâlnește un prototip de funcție pentru un șablon sau un apel de funcție definită printr-un șablon, el creează o funcție, substituind tipul specificat cu primul parametru al funcției. Fiecare din șabloanele examinate anterior a folosit un singur simbol de tip. În general, funcțiile lucrează cu diferite tipuri de parametri. Acest lucru este valabil și pentru funcțiile definite prin șabloane. Să considerăm, de exemplu, că programul trebuie să compare suma valorilor unui tablou cu o anumită valoare limită. Dacă tabloul conține numere întregi, funcția *sum\_and\_compare* ar putea avea următoarea formă:

```

int sum_and_compare(int *array, long value, int num_elements)
{
    long sum = 0;

```

```

for (int i = 0; i < num_elements; i++)
    sum += array[i];

return((sum > value) ? 1: 0);
}

```

Dacă tabloul ar conține valori reale, funcția ar putea arăta astfel:

```

int sum_and_compare(float *array, float value, int num_elements)
{
    float sum = 0;

    for (int i = 0; i < num_elements; i++)
        sum += array[i];

    return((sum > value) ? 1: 0);
}

```

După cum se observă, funcțiile folosesc tipuri diferite de parametri pentru tablou, ca și pentru variabila locală *sum*.

Pentru a crea un șablon pentru funcția *sum\_and\_compare*, acesta trebuie să admită două tipuri, cum se indică mai jos:

```

template<class T1, class T2>
int sum_and_compare(T1 *array, T2 value, int num_elements)
{
    T2 sum = 0;

    for (int i = 0; i < num_elements; i++)
        sum += array[i];

    return((sum > value) ? 1: 0);
}

```

În acest caz, C++ va folosi primii doi parametri specificați în prototipul funcției sau în apelul de funcție pentru a determina tipul șablonului. În situația când programul ar folosi un prototip de funcții, compilatorul C++ va substitui tipurile după cum se arată în continuare:

```

int sum_and_compare(int *array, long value, int num_elements);
//
int sum_and_compare(T1 *array, T2 value, int num_elements)
{
    T2 sum = 0;

    for (int i = 0; i < num_elements; i++)
        sum += array[i];
}

```

```

return((sum > value) ? 1: 0);
}
//
int sum_and_compare(int *array, long value, int num_elements)
{
    long sum = 0;

    for (int i = 0; i < num_elements; i++)
        sum += array[i];

    return((sum > value) ? 1: 0);
}

```

După cum se observă, compilatorul folosește tipul primului parametru pentru simbolul *T1* și tipul celui de al doilea parametru pentru simbolul *T2*. Următorul program *USE\_MULT.CPP*, folosește un șablon pentru a crea funcții ce admit tablouri de tip *int* sau *float*:

```

#include <iostream.h>

template<class T1, class T2>
int sum_and_compare(T1 *array, T2 value, int num_elements)
{
    T2 sum = 0;

    for (int i = 0; i < num_elements; i++)
        sum += array[i];

    return((sum > value) ? 1: 0);
}

int sum_and_compare(int *array, long value, int num_elements);
int sum_and_compare(float *array, float value, int num_elements);

void main(void)
{
    int values[] = { 1000, 20000, 30000, 4000, 500 };
    float prices[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };

    if (sum_and_compare(values, 32767L, 5))
        cout << "Values will overflow an int value" << endl;

    if (sum_and_compare(prices, 25.5, 5))
        cout << "Values exceed 25.5" << endl;
    else
        cout << "Values are less than 25.5" << endl;
}

```



### ȘABLOANE CARE ADMIT MAI MULTE TIPURI

La crearea șabloanelor de funcții, apar situații când trebuie folosite două sau mai multe tipuri distincte. În acest caz, definiția șablonului trebuie să specifice simbolurile pentru fiecare tip, precedate de cuvântul-cheie *template*, cum se arată mai jos:

```
template <class T1, class T2>
```

În acest caz, C++ va atribui simbolurilor *T1* și *T2* tipurile primilor doi parametri ai funcției, substituind ulterior tipurile în corpul funcției, acolo unde este necesar.

### UNDE SE PLASEAZĂ ȘABLOANELE

Șabloanele folosite într-un program se plasează într-un fișier antet care se include, ulterior, în program. În acest mod se reduce numărul de instrucțiuni din fișierul sursă al programului, facilitând citirea și înțelegerea programului. Dacă un programator dorește să examineze șabloanele, el poate deschide fișierul antet cu șabloane. În program, se recomandă folosirea prototipurilor de funcții, pentru a ajuta programatorii care citesc programul să înțeleagă mai bine folosirea șabloanelor.

### CREAREA ȘABLOANELOR DE CLASE

Așa cum sunt funcții care diferă numai prin tip, la fel există clase cu aceleași proprietăți. De exemplu, în Capitolul 5, ați creat o clasă *stack* care permite memorarea și recuperarea unor valori de tip *int*:

```
class stack {
public:
    stack(int size);
    int push(int value);
    int pop(void);
    int is_empty(void) { return(isempty); };
    int is_full(void) { return(isfull); };
private:
    int *storage; // Stack data buffer
    int elements; // Number of values in the stack
    int isempty; // True when stack is empty
    int isfull; // True when stack is full
    int stack_size; // Number of values the stack can store
};
```

Să presupunem acum că cerințele programului s-au schimbat și trebuie să memorăm valori de tip *int* și *float*. În loc de a crea două clase diferite, se poate proiecta un șablon de clasă, din care apoi se poate crea un obiect stivă de tipul *int* și un alt obiect stivă de tipul *float*.

Pentru a crea un șablon de clasă, se începe cu cuvântul-cheie *template*, urmat de simbolurile de clasă închise între paranteze unghiulare. În cadrul definiției clasei, simbolurile de tip clasă se vor folosi ca în exemplul următor:

```
template<class T> // Cuvânt cheie template și simbol de tip
class stack {
public:
    stack(int size);
    T push(T value); // Utilizarea simbolului de tip
    T pop(void);
    int is_empty(void) { return(isempty); };
    int is_full(void) { return(isfull); };
private:
    T *storage; // Stack data buffer
    int elements; // Number of values in the stack
    int isempty; // True when stack is empty
    int isfull; // True when stack is full
    int stack_size; // Number of values the stack can store
};
```

În cazul în care funcțiile membru sunt definite în afara clasei, ele trebuie precedate de cuvântul-cheie *template* și simbolurile de tip. De exemplu, funcțiile membru *push* și *pop* devin:

```
template<class T>
T stack<T>::pop(void)
{
    if (is_empty())
        return(0);
    else
    {
        if (--elements == 0)
            isempty = 1;

        isfull = 0;

        return(storage[elements]);
    }
}

template<class T>
T stack<T>::push(T value)
{
    if (is_full())
        return(0);
```

```

else
{
    isempty = 0;
    storage[elements++] = value;

    if (elements == stack_size)
        isfull = 1;

    return(value);
}
}

```

Pentru a crea un obiect *stack* în program, se va folosi numele clasei și al tipului în funcția *main*, ca mai jos:

```

void main(void)
{
    stack<int> values(10);
    stack<float> prices(10);

    // Other statements here
}

```

Nume clasă *stack*  
 Tip obiect  
 Dimensiune dorită a stivei

Următorul program, STACKEXP.CPP, creează și folosește două obiecte de tip *stack*. Unul dintre obiecte conține valori de tip *int*, iar celălalt valori de tip *float*:

```

#include <iostream.h>
#include <iomanip.h>

template<class T>
class stack {
public:
    stack(int size);
    T push(T value);
    T pop(void);
    int is_empty(void) { return(isempty); };
    int is_full(void) { return(isfull); };
private:
    T *storage;           // Stack data buffer
    int elements;         // Number of values in the stack
    int isempty;          // True when stack is empty
    int isfull;           // True when stack is full
    int stack_size;       // Number of values the stack can store
};

template<class T>

```

```

stack<T>::stack(int size)
{
    storage = new T[size]; // Allocate memory for the stack
    elements = 0;
    isempty = 1;
    isfull = 0;
    stack_size = size;
}

```

```

template<class T>
T stack<T>::pop(void)
{
    if (is_empty())
        return(0);
    else
    {
        if (--elements == 0)
            isempty = 1;

        isfull = 0;

        return(storage[elements]);
    }
}

```

```

template<class T>
T stack<T>::push(T value)
{
    if (is_full())
        return(0);
    else
    {
        isempty = 0;
        storage[elements++] = value;

        if (elements == stack_size)
            isfull = 1;

        return(value);
    }
}

```

```

void main(void)
{
    stack<int> values(10);
    stack<float> prices(15);

    for (int i = 0; ! values.is_full(); i++)

```

```

    values.push(i);

    while (! values.is_empty())
        cout << values.pop() << endl;

    for (i = 0; ! prices.is_full(); i++)
        prices.push(i * 100.0);

    while (! prices.is_empty())
        cout << setiosflags(ios::showpoint) << prices.pop() << endl;
}

```

În exemplul anterior, programul plasează 10 valori în stiva de întregi și 15 valori în stiva de valori reale. La compilarea și execuția programului, pe ecran vor fi afișate valorile din fiecare stivă în parte.

### ANALIZA UNUI ALT EXEMPLU

Următorul program, LINKLIST.CPP creează o clasă *LinkedList* ai cărei membri arată astfel:

```

struct Node {
    int value;
    Node *next;
    Node *previous;
};

class LinkedList {
public:
    LinkedList(void);
    void show_list(void);
    Node *append_value(int);
private:
    Node *first;
    Node *end;
};

```

După cum se observă, clasa ține evidența unei liste cu legături duble, între primul și ultimul său nod. Clasa folosește funcția *show\_list* pentru a afișa conținutul listei, și funcția *append\_value* pentru a insera valori la sfârșitul listei.

Următoarele instrucțiuni implementează întregul program LINKLIST.CPP:

```
#include <iostream.h>
```

```

struct Node {
    int value;
    Node *next;
    Node *previous;
};

```

```

}

class LinkedList {
public:
    LinkedList(void);
    void show_list(void);
    Node *append_value(int);
private:
    Node *first;
    Node *end;
};

LinkedList::LinkedList(void)
{
    first = NULL;
    end = NULL;
}

void LinkedList::show_list(void)
{
    Node *node;

    node = first;

    while (node)
    {
        cout << node->value << endl;
        node = node->next;
    }
}

Node *LinkedList::append_value(int value)
{
    Node *ptr = end;

    end = new Node;

    if (first == NULL)
        first = end;
    else
        ptr->next = end;

    if (end)
    {
        end->next = NULL;
        end->previous = ptr;
        end->value = value;
    }
}

```

```

    return(end);
}

void main(void)
{
    LinkedList list;

    for (int i = 0; i < 10; i++)
        list.append_value(i);

    list.show_list();

    for (i = 10; i < 20; i++)
        list.append_value(i);

    list.show_list();
}

```

În exemplul precedent, lista cu legături lucrează numai cu elemente de tip *int*. Folosind un șablon de clasă, puteți modifica rapid programul pentru a lucra cu valori de tip *int* și *float*. Următorul program, LINKTEMP.CPP, folosește un șablon pentru a crea două astfel de liste:

```

#include <iostream.h>
#include <iomanip.h>

template<class T>
struct Node {
    T value;
    Node *next;
    Node *previous;
};

template<class T1>
class LinkedList {
public:
    LinkedList(void);
    void show_list(void);
    Node<T1> *append_value(T1);
private:
    Node<T1> *first;
    Node<T1> *end;
};

template<class T1> LinkedList<T1>::LinkedList(void)
{
    first = NULL;
    end = NULL;
}

template<class T1> void LinkedList<T1>::show_list(void)

```

```

{
    Node<T1> *node;

    node = first;

    while (node)
    {
        cout << node->value << endl;
        node = node->next;
    }
}

template<class T1>
Node<T1> *LinkedList<T1>::append_value(T1 value)
{
    Node<T1> *ptr = end;

    end = new Node<T1>;

    if (first == NULL)
        first = end;
    else
        ptr->next = end;

    if (end)
    {
        end->next = NULL;
        end->previous = ptr;
        end->value = value;
    }

    return(end);
}

void main(void)
{
    LinkedList<int> list;
    LinkedList<float> values;

    for (int i = 0; i < 10; i++)
        list.append_value(i);

    list.show_list();

    cout << setiosflags(ios::showpoint);

    for (i = 1; i < 20; i++)
        values.append_value(i * 100.0);

    values.show_list();
}

```

Lista cu legături este alcătuită din noduri, fiecare nod conținând o valoare și pointeri la nodul precedent și nodul următor din listă, cum se arată în Figura 6.1.

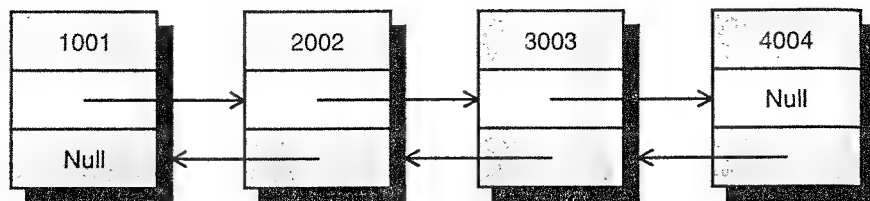


Figura 6.1. Structura de noduri în cadrul listei cu legături

Pentru a permite structurii de noduri să conțină valori de diferite tipuri, cum sunt *int* și *float*, programul ar trebui să folosească un șablon, cum este următorul:

```
template<class T>
struct Node {
    T value;
    Node *next;
    Node *previous;
};
```

Apoi, programul folosește un șablon de clasă ce definește operațiile pe o listă cu legături. Folosind acest șablon, clasa va putea fi folosită pentru a crea obiecte de tip listă cu legături ce conțin valori întregi, reale ș.a.m.d.

```
template<class T1>
class LinkedList {
public:
    LinkedList(void);
    void show_list(void);
    Node<T1> *append_value(T1);
private:
    Node<T1> *first;
    Node<T1> *end;
};
```

Programul va putea construi două obiecte de tip listă cu legături, una ce conține valori întregi și alta ce conține valori reale, folosind următoarele instrucțiuni:

```
LinkedList<int> list;
```

```
LinkedList<float> values;
```

Când compilatorul C++ va întâlni definițiile obiectelor, va substitui mai întâi simbolul de tip *T1* cu tipul actual. În cadrul clasei, compilatorul va crea structuri

de noduri, înlocuind simbolul *T* cu tipul care a substituit simbolul *T1*. În cazul unei liste de întregi, de exemplu, substituțiile vor fi:

```
LinkedList<int> list;
template<class T1>
class LinkedList {
public:
    LinkedList(void);
    void show_list(void);
    Node<T1> *append_value(T1);
private:
    Node<T1> *first;
    Node<T1> *end;
};
class LinkedList {
public:
    LinkedList(void);
    void show_list(void);
    Node *append_value(int);
private:
    Node<int> *first;
    Node<int> *end;
};
template<class T>
struct Node {
    T value;
    Node *next;
    Node *previous;
};
struct Node {
    int value;
    Node *next;
    Node *previous;
};
```

Așa cum se observă, compilatorul abordează asemenea șabloane într-o manieră pas cu pas. Când folosiți șabloane complexe în programe, încercați să realizați substituții similare pe instrucțiunile respective, pentru a înțelege mai bine folosirea șabloanelor.

## REZUMAT

Șabloanele C++ permit compilatorului să creeze în mod automat funcții și clase ale căror parametri sau membri diferă numai prin tip. Folosirea compila-



torului pentru a crea copii de funcții și clase despre care se știe că funcționează corect micșorează cantitatea de cod care trebuie scrisă și șansele de introducere a erorilor. Acest capitol a analizat șabloanele în detaliu. Înainte de trece la cu Capitolul 7, asigurați-vă că ați învățat următoarele:

- ✓ Un șablon este o schiță de funcție sau clasă pe care compilatorul C++ o va concretiza cu tipurile de parametri furnizate. Șabloanele sunt ideale pentru funcții sau clase apropiate între ele, ai căror parametri sau membri diferă numai prin tip.
- ✓ C++ acceptă șabloanele de funcții și șabloanele de clase.
- ✓ Pentru a crea un șablon de funcție, se folosește cuvântul-cheie *template* și simbolurile de tip între paranteze unghiulare, ca de exemplu *template <class T1>*. Restul declarației de funcție rămâne neschimbat, cu excepția faptului că se folosește simbolul T1 în locul unui nume de tip ca *int* sau *float*.
- ✓ Pentru ca ulterior să creați o funcție cu parametri de un anumit tip, includeți un prototip de funcție, sau folosiți funcția pur și simplu. În timpul compilării, C++ va determina corect tipurile pe care trebuie să le accepte, prin analiza tipurilor de parametri transmise funcției.
- ✓ Dacă șablonul de funcții acceptă tipuri distincte de parametri, compilatorul va atribui tipurile așa cum este specificat în prototipul funcției, sau de la stânga la dreapta, în funcție de parametrul transferat funcției.
- ✓ Pentru a crea un șablon de clasă, scrieți în fața definiției clasei cuvântul-cheie *template* urmat de simbolurile de clasă, după cum s-a arătat.
- ✓ Pentru a crea obiecte folosind șabloane de clasă, specificați numele clasei, urmat de tipul de obiect dorit scris între paranteze unghiulare, ca de exemplu *ClassName <int> ObjectName*.
- ✓ Dacă șablonul de clasă acceptă tipuri distincte de parametri, compilatorul va atribui tipurile așa cum este specificat în declarația obiectului, de la stânga spre dreapta. De exemplu, următoarea declarație folosește tipul *int* pentru primul simbol de tip și tipul *float* pentru al doilea simbol: *ClassName <int, float> ObjectName*.
- ✓ Dacă un șablon de clasă folosește al doilea șablon, compilatorul va efectua substituția tipurilor unul câte unul, începând cu primul tip de clasă menționat.

## CAPITOLUL 7

### ACOMODAREA CU ALOCAREA MEMORIEI

Când programele folosesc mari cantități de informații, ca de exemplu rezultatele unor teste la o clasă de 50 de elevi sau fișele individuale a 100 de angajați, sau chiar un șir de caractere foarte lung, ele pot memora aceste informații fie în tablouri, fie într-o zonă de memorie alocată dinamic. Avantajele alocării dinamice a memoriei constau în faptul că necesarul de memorie corespunde mai exact necesităților programului, ca ulterior, după terminarea prelucrării, aceasta să fie eliberată și disponibilizată în alte scopuri.

În limbajul C, funcțiile *alloc*, *calloc* și *malloc* din biblioteca de funcții permit alocarea dinamică a memoriei în timpul execuției. Când memoria nu mai este necesară, programele C o eliberează cu ajutorul funcției *free*. Deși programele C++ pot realiza alocarea dinamică a memoriei folosind aceste funcții de bibliotecă, se preferă folosirea operatorilor *new* și *delete*. Capitolul de față examinează acești operatori în detaliu. Veți învăța nu numai cum se alocă memorie pentru variabile de tip *int*, *float* sau structuri, ci și modul de alocare a memoriei pentru clasele C++. La sfârșitul acestui capitol, veți înțelege următoarele:

- ♦ Cum se alocă memorie în programele C++
- ♦ Ce se întâmplă când operatorul *new* nu reușește să aloce memorie
- ♦ Cum se poate specifica o funcție care se execută automat când o cerere de memorie nu poate fi satisfăcută
- ♦ Cum se eliberează memoria anterior alocată
- ♦ Cum se inițializează o locație de memorie cu o anumită valoare
- ♦ Cum se alocă memorie pentru un tablou de valori
- ♦ Cum se suprapun operatorii *new* și *delete*

#### UN EXEMPLU SIMPLU DE ALOCARE A MEMORIEI

Următorul program, SORT\_TEN.CPP, definește un tablou ce poate memora 10 valori întregi. Programul completează apoi tabloul cu 10 valori aleatoare, transferând apoi tabloul funcției *bubble\_sort* pentru sortare. În final se afișează valorile sortate:

```
#include <iostream.h>
#include <stdlib.h>

void bubble_sort(int *array, int num_elements)
{
    for (int i = 0; i < num_elements; i++)
        for (int j = 0; j < num_elements; j++)
            if (array[i] < array[j])
            {
                int temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
}

void main(void)
{
    int array[10];

    for (int i = 0; i < 10; i++)
        array[i] = random(SIZE); // Values from 0 through SIZE

    bubble_sort(array, 10);

    for (i = 0; i < 10; i++)
        cout << array[i] << endl;
}
```

**Observație:** Dacă compilatorul pe care îl folosiți nu acceptă funcția **random()**, atunci funcția **rand()** ar trebui să fie disponibilă. Cu toate că nu sunt identice, cele două funcții realizează acțiuni similare.

La compilarea și execuția acestui program, pe ecran vor apărea 10 valori sortate. Când numărul de valori cu care programul lucrează este fix, de exemplu 10 valori întregi, se pot folosi tablouri similare celui din programul precedent. Dar dacă tabloul este folosit numai la începutul sau la sfârșitul programului, atunci se consumă inutil memorie pentru o durată îndelungată de timp. În astfel de situații, memoria pentru tablouri trebuie alocată dinamic, după cerințele impuse.

Alocarea dinamică a memoriei reduce, de asemenea, numărul schimbărilor efectuate asupra programului, în cazul în care specificațiile acestuia se modifică. Să presupunem, de exemplu, că programul precedent trebuie să sorteze și să afișeze 20 de valori întregi. Pentru a schimba programul, trebuie înlocuită fiecare apariție a numărului 10 cu 20 și apoi trebuie recompilat programul.

O modalitate de a reduce numărul acestor schimbări este de a defini tabloul și ciclurile în funcție de o constantă, așa cum se arată în programul următor, SORT\_CON.CPP:

```
#define SIZE 20

void main(void)
{
    int array[SIZE];

    for (int i = 0; i < SIZE; i++)
        array[i] = random(SIZE); // Values from 0 through SIZE

    bubble_sort(array, SIZE);

    for (i = 0; i < SIZE; i++)
        cout << array[i] << endl;
}
```

**Observație:** Acest program nu conține codul pentru funcția **bubble\_sort** descrisă anterior.

În cazul când tabloul își va schimba dimensiunea, singura instrucțiune care trebuie modificată este definiția dimensiunii acestuia:

```
#define SIZE 20
```

În multe cazuri, nu se pot cunoaște cerințele de memorie înainte de execuția programului. Să presupunem că utilizatorul va furniza numărul de elemente ale tabloului. Dacă se folosesc tablouri, programul va trebui să aloce un tablou de dimensiune maximă și să folosească numai elementele necesare. Următorul program, BIGARRAY.CPP, realizează acest lucru:

```
#define SIZE 30000

void main(void)
{
    int array[SIZE];
    int size;

    cout << "Type in the number of values to sort (less than " <<
        SIZE << "): ";
    cin >> size;

    if (size <= SIZE)
    {
        for (int i = 0; i < size; i++)
            array[i] = random(SIZE); // Values from 0 through SIZE

        bubble_sort(array, size);

        for (i = 0; i < size; i++)
            cout << array[i] << endl;
    }
}
```

```
else
    cerr << "Array size must be less than " << SIZE << endl;
}
```

**Observație:** Acest program nu conține codul funcției `bubble_sort` descrisă anterior.

În acest exemplu, programul declară un tablou de 30 000 de valori. Dacă utilizatorul are nevoie de mai puține valori, o parte din tablou nu este folosită. Așa cum vom vedea, o soluție mai bună este de a alocă memoria dinamic.

**Observație:** Programul anterior a folosit valoarea 30 000 pentru dimensiunea tabloului. Această dimensiune nu a fost aleasă la întâmplare. Multe compilatoare pentru PC încă impun restricția ca tabloul să nu depășească 64K (deoarece tabloul este memorat într-un segment de 64 K). Cum un element întreg ocupă 2 octeți, atunci întregul tablou va conține  $30\,000 \times 2 = 60\text{K}$ , ceea ce se încadrează într-un segment, lăsând liber un mic spațiu pentru alte utilizări.

Următorul program, `C_MALLOC.CPP`, alocă memoria pentru tablou în mod dinamic, folosind funcția de bibliotecă `malloc`. În acest mod programul poate alocă un tablou a cărui dimensiune este în concordanță cu cerințele utilizatorului:

```
#include <malloc.h>

#define SIZE 30000

void main(void)
{
    int *array;
    int size;

    cout << "Type in the number of values to sort (less than " <<
        SIZE << "): ";
    cin >> size;

    if (size <= SIZE)
    {
        if ((array = (int *) malloc(size * sizeof(int))) == NULL)
            cerr << "Error allocating memory" << endl;
        else
        {
            for (int i = 0; i < size; i++)
                array[i] = random(SIZE); // Values from 0 through SIZE

            bubble_sort(array, size);

            for (i = 0; i < size; i++)
                cout << array[i] << endl;
        }
    }
}
```

```
free(array);
}
else
    cerr << "Array size must be less than " << SIZE << endl;
}
```

**Observație:** Acest program nu conține codul funcției `bubble_sort`.

Prototipul pentru funcția `malloc` se află în fișierul antet `MALLOC.H`. Funcția `malloc` returnează un pointer la regiunea de memorie alocată, sau valoarea `NULL` dacă cererea de memorie nu este satisfăcută.

```
if ((array = (int *) malloc(size * sizeof(int))) == NULL)
    cout << "Error allocating memory" << endl;
```

După cum se observă, programul apelează `malloc` cu numărul de octeți cerut, care în acest caz este numărul de elemente (`size`) înmulțit cu dimensiunea fiecărui element (`sizeof(int)`). După terminarea lucrului cu memoria alocată, programul o disponibilizează folosind funcția de bibliotecă `free`. În mod prestabilit, funcția `malloc` returnează un pointer de tip `void`. Simbolul (`int*`) care precede funcția convertește explicit pointerul de tip `void` la un pointer de tip `int`.

Acest program restricționează, de asemenea, dimensiunea tabloului, la mai puțin de 30 000. Ca și în cazul tablourilor statice, multe compilatoare ce folosesc sistemul DOS impun ca memoria dinamică alocată de un program să nu depășească un segment de 64 K. Veți învăța, puțin mai târziu, cum se determină cantitatea de memorie pe care programele o pot alocă dinamic.



#### LUCRUL CU MEMORIA DINAMICĂ

După cum se cunoaște, tablourile permit programelor să memoreze informații de același tip. De exemplu, un tablou ar putea conține rezultatele testelor a 50 de studenți, altul ar putea conține salariile a 100 de angajați. Deși tablourile sunt foarte convenabile pentru gruparea datelor, ele pot introduce necesitatea efectuării multor schimbări în program când numărul de elemente conținute trebuie să se modifice. În plus, dacă tablourile sunt folosite numai în anumite momente din cursul prelucrării datelor programului, ele consumă inutil memorie, chiar și când nu sunt folosite. Ca o alternativă, programele pot alocă memoria dinamic, pe măsura cerințelor. Memoria va fi astfel alocată în cantitatea necesară și la momentul necesar. La terminarea lucrului cu memoria, programul o poate elibera, disponibilizând-o pentru alte necesități. Operatorul `new` din C++ realizează alocarea dinamică a memoriei, iar operatorul `delete` disponibilizează memoria alocată anterior.

## ALOCAREA DINAMICĂ A MEMORIEI ÎN C++

Alocarea dinamică a memoriei în C++ se realizează cu ajutorul operatorilor *new* și *delete*, care sunt asemănători cu funcțiile de bibliotecă *malloc* și *free* discutate anterior. De exemplu, pentru a alocă dinamic un tablou de 100 de valori întregi, se folosește *new* astfel:

```
int *array = new int[100];
```

Ca și în cazul funcției *malloc*, dacă operatorul *new* nu poate alocă memoria cerută, el va atribui pointerului respectiv valoarea NULL. În program, testarea atribuirii valorii NULL unui pointer se poate face astfel:

```
if (array == NULL)
    cerr << "Unable to allocate memory" << endl;
```

**Observație:** Așa cum veți învăța în continuare, unele compilatoare nu atribuie valoarea NULL când operatorul *new* nu poate alocă memoria cerută. În schimb, unele compilatoare stopează imediat programul (prin apariția unei erori ce nu poate fi preluată, vezi Capitolul 14). Însă C++ vă permite să specificați o funcție proprie, care intră în execuție când memoria nu poate fi alocată. În acest fel, programele pot trata erorile de tip memorie insuficientă într-un mod personal.

La terminarea lucrului cu memoria, aceasta va fi eliberată astfel:

```
delete array;
```

Următorul program, NEW\_TEN.CPP, folosește operatorul *new* pentru a alocă un tablou de 10 valori întregi. Programul atribuie apoi tabloului valori aleatoare, după care îl sortează și îi tipărește elementele. Când tabloul nu mai este necesar în program, operatorul *delete* va elibera zona de memorie corespunzătoare:

```
void main(void)
{
    int *array = new int[10];

    if (array == NULL)
        cerr << "Error allocating memory" << endl;
    else
    {
        for (int i = 0; i < 10; i++)
            array[i] = random(30000); // Values from 0 thru 30,000

        bubble_sort(array, 10);

        for (i = 0; i < 10; i++)
            cout << array[i] << endl;
    }
}
```

```
delete array;
```

**Observație:** Acest program nu conține și codul funcției *bubble\_sort*.

După cum se observă, dacă operatorul *new* reușește să alocă memoria, programul va trata tabloul alocat dinamic exact ca și pe unul obișnuit. Următorul program, NEW\_ASK.CPP, cere utilizatorului să introducă volumul de memorie necesar tabloului. Programul va alocă apoi tabloul folosind operatorul *new*:

```
#define SIZE 30000

void main(void)
{
    int *array;
    int size;

    cout << "Type in the number of values to sort (less than " <<
        SIZE << "): ";
    cin >> size;

    if (size <= SIZE)
    {
        if ((array = new int[size]) == NULL)
            cerr << "Error allocating memory" << endl;
        else
        {
            for (int i = 0; i < size; i++)
                array[i] = random(SIZE); // Values from 0 thru SIZE

            bubble_sort(array, size);

            for (i = 0; i < size; i++)
                cout << array[i] << endl;


            delete array;
        }
    }
    else
        cerr << "Array size must be less than " << SIZE << endl;
}
```

**Observație:** Acest program nu conține și codul funcției *bubble\_sort*.

Dacă memoria nu poate fi alocată, atunci valoarea NULL va fi atribuită pointerului.

Se poate observa că programul folosește operatorul *new* pentru a alocă memoria cerută într-o instrucțiune *if*:

```
if ((array = new int[size]) == NULL)
    cerr << "Error allocating memory" << endl;
```



### ALOCAREA DINAMICĂ A MEMORIEI FOLOSIND OPERATORUL NEW

Operatorul *new* din C++ permite programului să aloce dinamic memoria. La folosirea operatorului *new* se specifică numărul de elemente cerute. Numărul total de octeți va putea fi calculat pe baza tipului de date pe care programul îl alocă. Pentru un element de tip *int*, de exemplu, operatorul *new* alocă doi octeți. Pentru tipul *float*, va alocă patru octeți. Dacă operatorul *new* reușește să aloce memoria, el va returna un pointer la zona de memorie alocată. Dacă operatorul *new* nu poate satisface cerința de memorie, el va returna valoarea NULL. Următoarea instrucțiune, de exemplu, folosește *new* pentru a alocă un tablou de 50 de elemente de tip *int*.

```
int *array = new int[50];
```

Tip element  
Pointer la memoria alocată

Număr elemente  
Tip element alocat

Analog, următoarea instrucțiune alocă un tablou de 100 elemente reale:

```
float *salaries = new float[100];
```

După ce programul alocă memorie cu operatorul *new*, trebuie testată valoarea pointerului pentru a ne asigura că operația a reușit. Dacă *new* nu poate alocă memoria cerută, atunci el va atribui pointerului valoarea NULL:

```
if (salaries == NULL)
    cerr << "Error allocating memory" << endl;
```

Când în program nu mai este nevoie de memoria alocată, ea trebuie eliberată cu operatorul *delete*, pentru a fi ulterior refolosită:

```
delete array;
delete salary;
```

Programul anterior a folosit operatorul *new* pentru alocarea memoriei necesare unui tablou de tip *int*. În același mod se pot alocă zone de memorie pentru tablouri care să conțină informații de alt tip. Următorul program, SUMFLOAT.CPP, folosește operatorul *new* pentru a alocă un tablou de 15 valori

reale. Programul atribuie apoi valori aleatoare elementelor tabloului, după care afișează aceste valori și suma lor:

```
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>

#define SIZE 15

void main(void)
{
    float *array = new float[SIZE];
    float sum = 0.0;

    if (array == NULL)
        cerr << "Error allocating memory" << endl;
    else
    {
        for (int i = 0; i < SIZE; i++)
        {
            array[i] = rand() / 1000.0;
            sum += array[i];
            cout << setw(6) << setprecision(2) <<
                setiosflags(ios::fixed || ios::right) <<
                array[i] << endl;
        }

        cout << "Array sum is " << sum << endl;
        delete array;
    }
}
```

După cum se observă, programul alocă memorie cu operatorul *new* și o eliberează cu operatorul *delete*.

**Observație:** Dacă memoria alocată nu se eliberează cu operatorul *delete*, ea va fi totuși automat eliberată la sfârșitul programului. Aveți, totuși, în vedere, că unul din motivele alocării dinamice este de a alocă memorie după necesități și de a o elibera imediat ce nu mai este utilă, pentru a fi refolosită în alte scopuri.

### LUCRUL CU VARIABLE POINTER

Exemplele anterioare au alocat memorie pentru tablouri folosind operatorul *new*. Deoarece C++ tratează numele de tablou ca pe un pointer, nu trebuie să vă faceți griji în privința variabilelor pointer și a redirectării pointerilor. Dacă în program alocăți memorie pentru altceva decât tablouri, atunci trebuie să folosiți pointeri pentru accesul la memoria alocată. De exemplu, următorul program USE\_PTRS.CPP, alocă memorie pentru valori de tip *int* și *float*, atribuie valori

acestor locații de memorie și afișează aceste valori referite prin intermediul pointerilor:

```
#include <iostream.h>

void main(void)
{
    int *int_pointer;
    float *float_pointer;

    if ((int_pointer = new int) == NULL)
        cerr << "Error allocating memory for int" << endl;
    else
    {
        *int_pointer = 1001;
        cout << "The value assigned to memory location " <<
            int_pointer << " is " << *int_pointer << endl;
    }

    if ((float_pointer = new float) == NULL)
        cerr << "Error allocating memory for float" << endl;
    else
    {
        *float_pointer = 11.22;
        cout << "The value assigned to memory location " <<
            float_pointer << " is " << *float_pointer << endl;
    }
}
```

După cum se observă, programul folosește operatorul de adresare indirectă când lucrează cu valori din locații de memorie referite prin pointeri. Observați instrucțiunea:

```
cout << "The value assigned to memory location " << float_pointer <<
    " is " << *float_pointer << endl;
```

Deoarece prima referință la *float\_pointer* nu este precedată de asterisc, programul va afișa valoarea acestui pointer, care este o adresă de memorie. A doua referință la *float\_pointer* este precedată de asterisc și, prin urmare, programul va afișa valoarea conținută în locația de memorie referită de *float\_pointer*. La compilarea și execuția acestui program, pe ecran va apărea o situație similară celei de mai jos:

```
C:\> USE_PTRS <ENTER>
The value assigned to memory location 0x14c0 is 1001
The value assigned to memory location 0x14c8 is 11.22
```

La compilarea și rularea programului locațiile de memorie pot să difere de la un sistem la altul, în funcție de sistemul de operare folosit, de programele rezidente în memorie sau de programele de control ale dispozitivelor I/O. După cum se poate vedea, programul afișează valoarea pointerului (adresa) și valoarea sto-

cată în locația de memorie. Figura 7.1 arată cum pointerii adresează locații de memorie ce conțin, la rândul lor, valori de diferite tipuri.

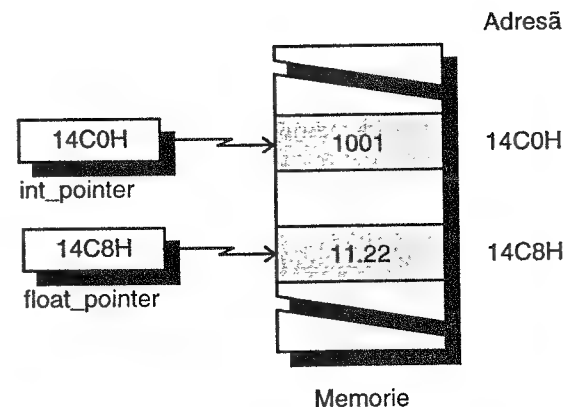


Figura 7.1. Un pointer conține adresa de memorie a unei valori sau adresa de început a unui domeniu de valori

Să observăm că programul anterior a declarat variabilele pointer astfel:

```
int *int_pointer;
float *float_pointer;
```

Dacă veți examina programe scrise în C sau C++, veți întâlni și declarații de pointeri care plasează asteriscul imediat după numele tipului acestora, ca mai jos:

```
int* int_pointer;
float* float_pointer;
```

De asemenea, puteți întâlni declarații la care asteriscul se află între numele tipului de pointer și numele variabilei:

```
int *int_pointer;
float *float_pointer;
```

Indiferent unde s-ar plasa asteriscul, toate cele trei forme creează variabile pointer identice. Pentru uniformitate, toate programele prezentate în această carte plasează asteriscul imediat înaintea numelui de variabilă.

### INIȚIALIZAREA UNEI VALORI FOLOSIND OPERATORUL NEW

La alocarea dinamică a unor variabile simple, ca cele de tipul *int* sau *float*, poate fi necesară atribuirea imediată a unor valori care să fie stocate în acele



locații de memorie. De exemplu, următoarele instrucțiuni alocă un pointer de tip *int* apoi atribuie valoarea 1001 locației de memorie respective:

```
int *int_ptr = new int;
if (int_ptr)
    *int_ptr = 1001;
```

Pentru mai multă comoditate, operatorul *new* permite specificarea unei valori inițiale în momentul alocării, plasând valoarea dorită între paranteze, ca mai jos:

```
int *int_ptr = new int(1001);
```

Dacă *new* reușește să aloce memorie, el va realiza și inițializarea. Dacă memoria nu poate fi alocată, inițializarea este ignorată. Următorul program, *NEW\_INIT.CPP*, folosește operatorul *new* pentru a aloca spațiu de memorie și apoi pentru a-l inițializa cu valori de tip *int* sau *float*.

```
#include <iostream.h>

void main(void)
{
    int *int_pointer = new int(1001);
    float *float_pointer = new float(3.12345);

    cout << *int_pointer << endl;
    cout << *float_pointer << endl;
}
```

### SĂ ÎNȚELEGEM ARITMETICA POINTERILOR

Mulți programatori în C sau C++ folosesc pointeri în loc de operații cu tablouri, pentru îmbunătățirea performanței programelor. Să considerăm, de exemplu, următoarele funcții, care afișează valorile într-un tablou de tip *float*:

<pre>void show_values(float array[], int size) {     for (int i = 0; i &lt; size; i++)         cout &lt;&lt; array[i] &lt;&lt; endl; }</pre>	<pre>void show_values(float *array, int size) {     for (int i = 0; i &lt; size; i++)         cout &lt;&lt; *array++; }</pre>
--	---

Funcția din stânga afișează valorile folosind un tablou și operații de indexare. Funcția din partea dreaptă, folosește un pointer pentru afișarea valorilor. Dacă includeți cele două funcții în programe diferite și generați listingurile în limbaj de asamblare, veți vedea că, în cazul folosirii tabloului, compilatorul va insera mai multe instrucțiuni decât în cazul pointerului. Drept urmare, implementarea cu pointeri conține mai puține instrucțiuni și este mai rapidă.

Într-un program, un pointer indică o valoare de un anumit tip. Singura excepție la această regulă sunt pointerii *void*, care nu au tip. Motivul pentru care pointerii

au tipuri asociate este de a cunoaște cu câți octeți trebuie incrementat pointerul în memorie când se dorește traversarea unei liste de valori ca cea anterioară. De exemplu, dacă folosiți un pointer pentru a traversa un șir de caractere, compilatorul va incrementa conținutul pointerului cu 1 la întâlnirea unei expresii de forma *string++*. Dacă folosiți un pointer la un tablou de întregi, atunci compilatorul va incrementa pointerul cu doi octeți pentru a adresa corect următorul element din tablou. Analog, pointerul către o listă de valori reale va fi incrementat cu 4. Dacă pointerii nu ar avea tipuri asociate, compilatorul nu ar putea realiza astfel de operații.

### TRATAREA POINTERILOR CA TABLOURI

Multe programe C++ tratează tablourile drept pointeri. De exemplu, următorul program, *SHOW\_STR.CPP*, transferă un tablou șir de caractere funcției *show\_string* ce afișează, la rândul ei, câte un caracter din șir folosind un pointer:

```
#include <iostream.h>

void show_string(char *string)
{
    while (*string)
        cout.put(*string++);
}

void main(void)
{
    char book[] = "Success with C++\n";

    show_string(book);
}
```

Dacă examinați programe în C sau C++, veți întâlni periodic coduri în care se combină operațiile cu tablouri și operațiile cu pointeri. De exemplu, următorul fragment de program atribuie valoarea 3 celui de al patrulea element (de indice 3) dintr-un tablou de întregi:

```
void some_function(int *array, int size)
{
    array[3] = 3;

    // Other statements
}
```

Se observă că funcția primește ca parametru un pointer la tipul *int*, după care tratează pointerul ca fiind adresa de început a unui tablou. Este important de reținut că orice pointer, în orice moment, poate fi tratat ca baza unui tablou.

Următorul program, PTRARRAY.CPP, realizează exact opusul operațiilor din programul precedent. În acest caz, funcția *show\_string* primește un pointer la un șir și tratează pointerul ca fiind baza unui tablou:

```
#include <iostream.h>

void show_string(char string[])
{
    for (int i = 0; string[i]; i++)
        cout.put(string[i]);
}

void main(void)
{
    char *book = "Hi, Hello, world!\n";

    show_string(book + 4);
}
```

Să observăm că programul transferă funcției adresa de început a șirului plus 4 (se omit caracterele "Hi,"). Dacă rulați acest program pe ecran va apărea:

```
C:\> PTRARRAY <ENTER>
Hello, world!
```

După cum se poate vedea, orice pointer poate fi tratat în orice moment ca fiind baza unui tablou.

### FOLOSIREA POINTERILOR CU PARAMETRII DE FUNCȚII

Una din cele mai folosite operații cu pointeri este schimbarea parametrilor unei funcții. În mod prestabilit, la apelarea unei funcții, C++ face copii ale valorilor parametrilor, punând aceste copii în stivă. Deoarece funcția primește o copie a valorii parametrului, ea nu poate schimba valoarea acestuia. Următorul program, NOCHANGE.CPP, transferă 3 parametri funcției *one\_two\_three*, care, la rândul ei, atribuie parametrilor valorile 1,2 și 3. Deoarece funcția nu are acces la parametrii efectiv, ci la niște copii ale acestora, parametrii rămân neschimbați după apelul funcției:

```
#include <iostream.h>

void one_two_three(int a, int b, int c)
{
    cout << "Values passed to the function: " << a << ' ' << b <<
        ' ' << c << endl;

    a = 1;
    b = 2;
    c = 3;
    cout << "Values within the function: " << a << ' ' << b <<
        ' ' << c << endl;
}
```

```
void main(void)
{
    int one = 0, two = 0, three = 0;
    one_two_three(one, two, three);
    cout << "Values in main: " << one << ' ' << two << ' ' <<
        three << endl;
}
```

La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> NOCHANGE <ENTER>
Values passed to the function: 0 0 0
Values within the function: 1 2 3
Values in main: 0 0 0
```

După cum se observă, valorile parametrilor actuali nu sunt modificate de funcție. Pentru a putea schimba valorile parametrilor, programul trebuie să folosească pointeri. Parametrii funcției *one\_two\_three* trebuie să fie pointeri, cum se arată mai jos:

```
void one_two_three(int *a, int *b, int *c)
{
    cout << "Values passed to the function: " << *a << ' ' <<
        *b << ' ' << *c << endl;

    *a = 1;
    *b = 2;
    *c = 3;
    cout << "Values within the function: " << *a << ' ' << *b <<
        ' ' << *c << endl;
}
```

Parametrii funcției precedente sunt pointeri la valori de tip *int*. La apelul funcției, trebuie transferată adresa fiecărei variabile folosind operatorul de adresare (&), cum se arată în continuare:

```
one_two_three(&one, &two, &three);
```

Transferând funcției adresa variabilelor, aceasta va lucra cu valorile efective ale parametrilor, în loc de copiile acestor valori. În acest fel, schimbările efectuate asupra parametrilor rămân în vigoare și după apelul funcției. Următorul program, PAR\_CHG.CPP, folosește pointeri pentru a schimba valorile parametrilor în funcția *one\_two\_three*:

```
#include <iostream.h>

void one_two_three(int *a, int *b, int *c)
{
    cout << "Values passed to the function: " << *a << ' ' <<
        *b << ' ' << *c << endl;

    *a = 1;
```



```
*b = 2;
*c = 3;
cout << "Values within the function: " << *a << " " << *b <<
    " " << *c << endl;
}
```

```
void main(void)
{
    int one = 0, two = 0, three = 0;

    one_two_three(&one, &two, &three);

    cout << "Values in main: " << one << " " << two << " " <<
        three << endl;
}
```

La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> PAR_CHG <ENTER>
Values passed to the function: 0 0 0
Values within the function: 1 2 3
Values in main: 1 2 3
```

Se observă că valorile transferate funcției se schimbă după apelul funcției. În Capitolul 10 se vor examina referințele C++, care oferă o a doua posibilitate de a schimba valorile parametrilor într-o funcție.

### FOLOSIREA POINTERILOR LA STRUCTURI ȘI CLASE

Așa cum se folosesc pointeri la variabile de tip *int*, *float*, *char* sau la tablouri de astfel de variabile, se pot folosi și pointeri la structuri sau clase. Se știe că, atunci când se lucrează cu structuri, se folosește operatorul punct (.) pentru a face referință la membrii structurii. De exemplu, următoarele instrucțiuni atribuie data de 25 decembrie 1994 structurii de tip *Date*:

```
struct Date {
    int month;
    int day;
    int year;
} xmas;

xmas.month = 12;
xmas.day = 25;
xmas.year = 1994;
```

Când lucrăm în program cu pointeri la structuri sau la clase, accesul la membri se poate face în unul din următoarele două moduri. Mai întâi, se poate folosi operatorul de indirectare și pointerul între paranteze. De exemplu, următoarea funcție, *assign\_xmas*, atribuie data unei structuri indicată prin variabila *date\_ptr*:

```
void assign_xmas(struct Date *date_ptr)
{
    (*date_ptr).month = 12;
    (*date_ptr).day = 25;
    (*date_ptr).year = 1994;
}
```

Deoarece această sintaxă poate încurca programatorii începători și deoarece programatorii uită sistematic parantezele, limbajele C și C++ oferă o a doua alternativă de a avea acces la membrii unei structuri indicați prin pointer sub forma următoare:

```
void assign_xmas(struct Date *date_ptr)
{
    date_ptr->month = 12;
    date_ptr->day = 25;
    date_ptr->year = 1994;
}
```

În cazul în care se lucrează cu clasele, aceleași două modalități pot fi folosite pentru accesul la membrii clasei.

### POINTERRII NU SUNT REFERINȚE

În Capitolul 10 veți învăța că o *referință* C++ este un nume asociat, sau un al doilea nume pe care programele îl pot folosi în cazul unei variabile. În general, referințele sunt destinate simplificării funcțiilor care schimbă valorile parametrilor, eliminând necesitatea operațiilor cu pointeri. De exemplu, următorul program, *SWAP\_PTR.CPP*, folosește pointeri pentru a modifica valorile variabilelor *a* și *b*:

```
#include <iostream.h>

void swap_values(int *first, int *second)
{
    int temp = *first;
    *first = *second;
    *second = temp;
}

void main(void)
{
    int a = 1;
    int b = 1001;

    swap_values(&a, &b);

    cout << "Values after swap a: " << a << " b: " << b << endl;
}
```

Analog, următorul program, SWAP\_REF.CPP, folosește referințe pentru schimbarea valorilor parametrilor:

```
#include <iostream.h>

void swap_values(int& first, int& second)
{
    int temp = first;
    first = second;
    second = temp;
}

void main(void)
{
    int a = 1;
    int b = 1001;

    int& a_alias = a;    // Declare the references
    int& b_alias = b;

    swap_values(a_alias, b_alias);

    cout << "Values after swap a: " << a << " b: " << b << endl;
}
```

Se observă că referințele elimină necesitatea folosirii operațiilor cu pointeri într-o funcție. Dar utilizarea referințelor mărește complexitatea programelor, prin introducerea unor variabile suplimentare. Capitolul 10 va examina referințele în detaliu. Deocamdată, rețineți că o referință nu este un pointer.

### ATENȚIE LA POINTERI ȘI LA VARIABLE LOCALE

Pe măsură ce programele devin mai complicate, apar situații când memoria trebuie alocată din interiorul funcțiilor. Este important să reținem că, atunci când memoria este alocată din interiorul unei funcții, ea rămâne alocată și după ce funcția se încheie. Să considerăm, de exemplu, programul LOCALPTR.CPP, care alocă 1000 de octeți în cadrul funcției *bad\_pointer*.

```
#include <iostream.h>

void bad_pointer(void)
{
    char *ptr = new char[1000];    // Allocate the memory

    // Perform some operation
}

void main(void)
{
    // Invoke function
}
```

```
bad_pointer():
{
    // Perform some operation

    // Invoke the function a second time
    bad_pointer();
}
```

La fiecare apel al funcției *bad\_pointer*, această alocă memorie, atribuind adresa zonei alocate variabilei locale *ptr*. Din păcate, funcția nu eliberează memoria adresată atunci când nu mai este nevoie de aceasta. Deoarece memoria este adresată folosind o variabilă locală, programul nu are nici o posibilitate de a disponibiliza memoria după terminarea funcției. Prin urmare, memoria este alocată, dar programul nu o poate accesa. Dacă într-un program se alocă temporar memorie din interiorul unei funcții, trebuie să ne asigurăm că el va elibera memoria înainte ca funcția să se încheie. Există însă și situații, cum se va vedea în continuare, când se dorește ca memoria să rămână alocată și după încheierea funcției.

### FUNCȚII CARE RETURNEAZĂ POINTERI

În funcție de destinația unei funcții, există situații când memoria alocată într-o funcție trebuie să rămână alocată și după încheierea funcției. De exemplu, următorul program, STR\_DUP.CPP, folosește funcția *string\_duplicate*, care alocă memorie pentru a reține caracterele dintr-un parametru de tip șir de caractere, returnând apoi un pointer către începutul noului șir. Dacă funcția nu reușește să copieze șirul, funcția va returna NULL.

```
#include <iostream.h>
#include <string.h>

char *string_duplicate(char *source)
{
    char *target = new char[strlen(source)];

    if (target)
        for (int i = 0; target[i] = source[i]; i++)
            ;

    return(target);
}

void main(void)
{
    char *new_string;

    new_string = string_duplicate("Success with C++!");

    cout << new_string << endl;
}
```

Deoarece programul trebuie să aibă acces la zona de memorie alocată de funcție și după încheierea acesteia, funcția nu eliberează memoria cu operatorul `delete`.

## SPAȚIUL LIBER DE MEMORIE

Memoria este alocată dintr-o zonă rezervată de memorie, denumită *spațiul liber* sau *zonă fragmentată* (heap). Așa cum se arată în Figura 7.2, spațiul liber este situat deasupra datelor programului și sub stiva programului.

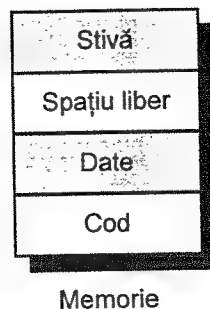


Fig. 7.2. Spațiul liber este situat între datele și stiva programului.

În funcție de sistemul de operare folosit și de modelul de memorie a compilatorului, cantitatea de spațiu liber disponibil va diferi de la un sistem la altul. Capitolul 15 va analiza în detaliu spațiul liber de memorie. Veți învăța ce se întâmplă la nivelul sistemului când se alocă sau se eliberează memorie. Deocamdată, însă, rețineți că programele nu au la dispoziție o cantitate nelimitată de memorie liberă. Este posibil ca unele programe să consume tot spațiul de memorie disponibilă.

## CONDIȚIILE DE MEMORIE INSUFICIENTĂ

Așa cum ați învățat, când operatorul `new` nu poate alocă cantitatea de memorie necesară, el atribuie valoarea `NULL` variabilei pointer corespunzătoare. Următorul program, `TAKE_ALL.CPP`, folosește operatorul `new` într-un ciclu pentru a alocă blocuri a câte 10 K de memorie, până când nu mai rămâne memorie disponibilă. Programul afișează apoi un mesaj indicând cantitatea de memorie alocată în total:

```
#include <iostream.h>

void main(void)
```

INFORMATICĂ

```
char *buffer;
long sum = 0L;

do {
    buffer = new char[10000];
    if (buffer)
    {
        cout << "Allocated 10,000 bytes" << endl;
        sum += 10000;
    }
} while (buffer);

cout << "The amount of memory allocated was " << sum <<
      " bytes" << endl;
}
```

Compilați și executați acest program. În majoritatea cazurilor, operatorul `new` va atribui valoarea `NULL` pointerului `buffer`. Unele compilatoare, însă, vor stopa programul când memoria cerută nu poate fi alocată, programul afișând rezultate de forma:

```
C:\> TAKE_ALL <ENTER>
Allocated 10,000 bytes
Allocated 10,000 bytes
Allocated 10,000 bytes
Allocated 10,000 bytes
Allocated 10,000 bytes
Abnormal program termination
```

Pentru a înțelege de ce programul se oprește brusc în acest mod, trebuie să știm că C++ permite specificarea unei funcții care urmează a fi executată când cererea de memorie nu poate fi satisfăcută. O astfel de funcție este `set_new_handler`, al cărui prototip se află definit în fișierul antet `NEW.H`.

Următorul program, `SET_NEW.CPP`, folosește `set_new_handler` pentru a apela funcția `message_and_exit` când cererea de memorie nu poate fi satisfăcută. Când funcția `message_and_exit` este apelată, se afișează un mesaj la fluxul `cerr` și programul este încheiat.

```
#include <iostream.h>
#include <stdlib.h>
#include <new.h>

void message_and_exit(void)
{
    cerr << "\aNo more memory!" << endl;
    exit(1);
}
```

```

void main(void)
{
    char *buffer;
    long sum = 0L;

    set_new_handler(message_and_exit);

    do {
        buffer = new char[10000];
        if (buffer)
        {
            cout << "Allocated 10,000 bytes" << endl;
            sum += 10000;
        }
    } while (buffer);

    cout << "The amount of memory allocated was " << sum <<
        " bytes" << endl;
}

```

**Observație:** Numele exact și sintaxa folosită de alte compilatoare pentru funcția `set_new_handler` poate fi diferită de cea indicată aici.

În marea majoritate a cazurilor de cerere nesatisfăcută de memorie, nu se doarește lansarea în execuție a unei funcții, ci pur și simplu returnarea rezultatului NULL de către operatorul `new`. În astfel de situații se poate invoca funcția `set_new_handler` cu parametrul NULL pentru dezactivarea funcției. Următorul program, `USE_NULL.CPP`, folosește acest apel:

```

#include <iostream.h>
#include <stdlib.h>
#include <new.h>

void main(void)
{
    char *buffer;
    long sum = 0L;

    set_new_handler(NULL);

    do {
        buffer = new char[10000];
        if (buffer)
        {
            cout << "Allocated 10,000 bytes" << endl;
            sum += 10000;
        }
    }
}

```

```

    }
} while (buffer);

cout << "The amount of memory allocated was " << sum <<
    " bytes" << endl;
}

```

Așa cum veți vedea în secțiunea următoare, cea mai bună metodă de a controla operațiile de alocare a memoriei este suprapunerea operatorilor `new` și `delete`.



### SPECIFICAREA UNEI FUNCȚII PENTRU TRATAREA ERORILOR DE MEMORIE INSUFICIENTĂ

În mod normal, dacă operatorul `new` nu poate satisface o cerere de memorie, se va atribui valoarea NULL pointerului corespunzător. În anumite situații (de exemplu, atunci când se testează programul) se poate dori afișarea unui mesaj de eroare la cerere de memorie nesatisfăcută. Terminând programul în acest mod, se poate localiza o operație de alocare a memoriei al cărei rezultat nu este verificat la NULL. Pentru a specifica o funcție ce urmează să fie executată la cerere de memorie nesatisfăcută, se poate folosi funcția `set_new_handler`. Fișierul antet `NEW.H` furnizează următorul prototip:

```
set_new_handler(* new_handler());
```

Pentru a invalida apelul unei funcții speciale de tratare a erorilor, se poate realiza un apel cu parametrul NULL:

```
set_new_handler(NULL);
```

### SUPRAPUNEREA OPERATORILOR NEW ȘI DELETE

În Capitolul 5 ați învățat cum se suprapun operatorii în program. Deoarece `new` și `delete` sunt operatori C++, și ei pot fi suprapuși. Așa cum veți vedea în Capitolul 15, C++ nu realizează așa-numita *colectare a resturilor* pentru gruparea spațiilor fragmentate de memorie într-o zonă mai mare, în scopul satisfacerii cerințelor de memorie. De exemplu, în Figura 7.3, spațiul liber conține memorie alocată și disponibilă. Se observă că există 500 de octeți de memorie liberă. Din păcate, cel mai mare bloc de memorie disponibilă are 250 de octeți. Dacă programul trebuie să aloce 300 de octeți, alocarea va eșua.

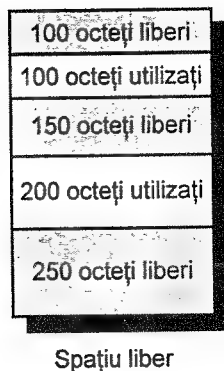


Figura 7.3. Locațiile fragmentate de spațiu liber împiedică alocarea memoriei.

**Colectarea resturilor** este o tehnică folosită pentru a grupa memoria disponibilă într-un bloc compact. Folosind tehnicile de colectare a resturilor, spațiul liber disponibil poate fi grupat ca în Figura 7.4.

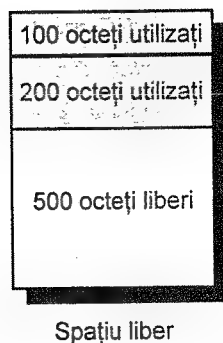


Figura 7.4. Gruparea spațiului liber disponibil.

Pentru a realiza colectarea resturilor, blocurile de memorie alocate și disponibile sunt deplasate pentru a crea un bloc mare de memorie disponibilă. C++ nu poate realiza colectarea de resturi, deoarece nu cunoaște cum sunt folosiți pointerii de către program în diferite regiuni de memorie. Totuși, programul poate realiza astfel de operații, deoarece are posibilitatea de a ține o evidență a folosirii pointerilor:

Un mod de a implementa colectarea resturilor este de a suprapune operatorii *new* și *delete* cu funcții care realizează periodic astfel de operații.

Din păcate, pașii ce trebuie realizați pentru a implementa o colectare de resturi eficientă și utilă depășește cadrul discuției din acest paragraf. Dar suprapunerea operatorilor *new* și *delete* nu este o chestiune complicată. Pentru aceasta, trebuie declarate funcții asemănătoare celor ce urmează:

```
void *operator new(size_t size)
{
    // Statements
}

void *operator new[](size_t size)
{
    // Statements
}

static void operator delete(void *pointer)
{
    // Statements
}
```

Se poate observa că majoritatea compilatoarelor furnizează două definiții pentru operatorul *new*, una care alocă memorie pentru tipuri simple de date (*int*, *float*), și alta care alocă memoria pentru tablouri. În mod normal, în cadrul operatorului *new* se alocă memorie cu funcția *malloc*, ca mai jos:

```
void *operator new(size_t size)
{
    // Statements
    return(malloc(size));
}
```

În interiorul operatorului *delete*, memoria se va elibera cu funcția *free*:

```
static void operator delete(void *pointer)
{
    // Statements
    free(pointer);
}
```

Suprapunerea operatorilor *new* și *delete* cere o oarecare experiență, în funcție de compilatorul folosit. De exemplu, următorul program, LIM\_1000.CPP, suprapune operatorul *new* pentru a nu alocă decât blocuri ce nu depășesc 1000 de octeți. În cazul când programul apelează *new* cu o valoare mai mare decât 1000, acesta va returna imediat valoarea NULL:

```
#include <iostream.h>
#include <alloc.h>
```

```
void *operator new[](size_t size)
{
    if (size >= 1000)
        return(0);

    return(malloc(size));
}

void main(void)
{
    char *string = new char[1000];
    float *salaries = new float[200];
    int *ages = new int[250];

    if (string)
        cout << "string[1000] was allocated" << endl;
    else
        cout << "Error allocating string[1000]" << endl;

    if (salaries)
        cout << "salaries[200] was allocated" << endl;
    else
        cout << "Error allocating salaries[200]" << endl;

    if (ages)
        cout << "ages[250] was allocated" << endl;
    else
        cout << "Error allocating ages[250]" << endl;
}
```

**Observație:** Alte compilatoare pot folosi fișierul `antet malloc.h` în loc de `alloc.h`.

În exemplul precedent, programul suprapune operatorul `new [ ]`. După cum se observă, programul încearcă să aloce memorie pentru diferite tipuri de tablouri, ale căror necesități de memorie sunt 1000, 8000 (4\*200 octeți) și 500 (2\*250 octeți). La compilarea și execuția programului, pe ecran se va afișa:

```
C:\> LIM_1000 <ENTER>
Error allocating string[1000]
salaries[200] was allocated
ages[250] was allocated
```

În mod normal, programele nu vor ignora operatorii globali `new` și `delete`. În schimb, programele vor ignora operatorii pentru un anumit tip de clasă. Următorul program, `OVER_NEW.CPP`, suprapune operatorul `new` pentru a fi folosit cu clasa `Date` în scopul menținerii în buffer-ul de date a valorii prestabilite 1 ianuarie 1994. O prelucrare similară s-ar putea realiza, de asemenea, pentru a încărca data curentă în buffer:

```
#include <iostream.h>
#include <alloc.h>

class Date {
public:
    void *operator new(size_t size);
    int month;
    int day;
    int year;
};

class DateClass {
public:
    DateClass(int month, int day, int year);
    void show_date(void);
private:
    Date *buffer;
};

DateClass::DateClass(int month, int day, int year)
{
    buffer = new Date;

    cout << "Default date: ";
    show_date();

    buffer->month = month;
    buffer->day = day;
    buffer->year = year;
}

void *Date::operator new(size_t size)
{
    Date *ptr = (Date *) malloc(size);

    if (ptr)
    {
        ptr->month = 1;
        ptr->day = 1;
        ptr->year = 1994;
    }

    return((void *) ptr);
}

void DateClass::show_date(void)
{
    cout << buffer->month << '/' << buffer->day << '/' <<
        buffer->year << endl;
}
```

```
void main(void)
{
    DateClass birthday(9, 30, 94);

    cout << "Birthday date: ";
    birthday.show_date();
}
```

**Observație:** Alte compilatoare pot folosi fișierul `antet malloc.h` în loc de `alloc.h`.

La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> OVER_NEW <ENTER>
Default date: 1/1/1994
Birthday date: 9/30/94
```

Se observă că, prin suprapunerea operatorilor *new* și *delete*, programele pot câștiga enorm în controlul operațiilor de alocare a memoriei.

## REZUMAT

Așa cum ați învățat, alocarea dinamică a memoriei are o serie de avantaje. În primul rând, programele pot determina mai precis necesarul de memorie în timpul execuției. În plus, alocarea dinamică a memoriei, în locul folosirii tablourilor, reduce numărul de schimbări efectuate în program în cazul modificării cerințelor de memorie. În sfârșit, alocarea memoriei atunci când este nevoie și eliberarea acesteia imediat ce nu mai este necesară permit ca memoria să fie folosită în alte scopuri. În Capitolul 8 veți învăța cum să folosiți funcții virtuale în cadrul programelor C++. Înainte de a trece la Capitolul 8, asigurați-vă că ați învățat următoarele:

- ✓ În limbajul C, pentru alocarea memoriei se folosesc funcțiile de bibliotecă *malloc* și *calloc*, iar pentru eliberarea memoriei se folosește funcția *free*. În C++, programele alocă memorie folosind operatorul *new* și o eliberează ulterior folosind operatorul *delete*.
- ✓ Dacă operatorul *new* nu poate alocă memorie, atunci va atribui valoarea NULL pointerului corespunzător. Unele compilatoare pot însă să apeleze o funcție care să încheie execuția programului.
- ✓ Folosind funcția *set\_new\_handler* se poate specifica o funcție care se execută automat când o cerere de memorie nu poate fi satisfăcută. În funcție de destinația programului, această funcție ar putea încerca să elibereze memoria astfel încât cererea să fie repetată. Dacă se apelează funcția *set\_new\_handler* cu parametrul NULL, atunci nu se va executa nici o funcție de tratare a erorilor de memorie insuficientă.

- ✓ Când nu mai este nevoie de memoria alocată anterior, programele o pot elibera folosind operatorul *delete*. La terminarea unui program, toată memoria alocată anterior va fi eliberată automat.
- ✓ Pentru a alocă memorie pentru o variabilă simplă, ca *int* sau *float*, se folosește operatorul *new* astfel:

```
int *int_pointer = new int;
float *float_pointer = new float;
```

- ✓ Dacă în operatorul *new* se include între paranteze rotunde o valoare, atunci locația de memorie alocată este inițializată cu această valoare (în cazul când cererea de memorie a fost satisfăcută):

```
int *int_pointer = new int(1001);
```

- ✓ Pentru a alocă memorie pentru un tablou de valori folosind operatorul *new*, se specifică numărul de elemente între paranteze pătrate, ca mai jos:

```
int *int_array = new int[50];
```

- ✓ În funcție de scopul unui program, pot apărea situații când se dorește ignorarea operatorilor *new* și *delete*. În mod normal, programele nu vor ignora operatorii globali, ci mai degrabă operatorii folosiți de o anumită clasă.

## CAPITOLUL 8

### ACOMODAREA CU FUNCȚIILE VIRTUALE ȘI POLIMORFISMUL

Este greu de găsit o carte sau un articol de revistă de C++ în care să nu se discute problema *polimorfismului*. Din păcate, în majoritatea cazurilor polimorfismul și folosirea funcțiilor virtuale par un lucru dificil. Așa cum veți învăța în acest capitol, partea cea mai dificilă a polimorfismului este termenul însuși. O dată ce treceți de acesta, conceptele sunt mai simple. Așa că, să exemplificăm termenul chiar acum. *Poly*, în limba greacă, înseamnă multe, iar *morfic*, înseamnă forme. Când combinați cei doi termeni, obțineți *multe forme*. Pe scurt, un obiect polimorfic este un obiect capabil de a avea două sau mai multe forme. De exemplu, să presupunem că avem un obiect de tip telefon. Pentru a realiza o convorbire, veți folosi funcția membru *dial* (formează număr). Așa cum se știe, un telefon poate folosi fie o claviatură cu butoane, fie un disc rotativ. Cu alte cuvinte, telefoanele pot avea una din cele două *forme*. În funcție de forma folosită, un obiect *telefon* va folosi o funcție membru *dial* diferită pentru formarea numărului. Acest capitol explică ce este și ce nu este polimorfismul. Veți învăța:

- ♦ Ce este polimorfismul și la ce folosește
- ♦ Cum suportă C++ polimorfismul
- ♦ Ce este o funcție virtuală și cum se definește
- ♦ Cum se înlocuiește o funcție virtuală cu o funcție proprie
- ♦ În ce condiții are loc polimorfismul
- ♦ Ce este o funcție pur virtuală

În Capitolul 13 veți analiza mai în detaliu ce înseamnă polimorfismul pentru compilatorul C++.

#### SĂ ÎNȚELEM FUNCȚIILE VIRTUALE

C++ implementează polimorfismul folosind *funcțiile virtuale*. În sensul cel mai simplu, o funcție virtuală este o funcție membru proiectată pentru a funcționa cu orice membru al clasei de bază sau al unei clase derivate (al cărui tip poate fi necunoscut). În acest fel, clasele pot alege să folosească funcția virtuală sau o altă funcție proprie, scutind astfel programatorii de a scrie diferite funcții pentru diferite clase. O funcție virtuală poate fi închipuită ca o "funcție variabilă", adică o funcție membru de clasă concepută pentru a fi înlocuită ulterior cu o funcție membru necunoscută a unei clase derivate. Prin definirea unei funcții drept *vir-*



*tuală*, un program poate crea obiecte cu propriile funcții membru, folosite în locul funcției virtuale, cât și obiecte care folosesc funcția prestabilită.

Declarația unei funcții virtuale este precedată de cuvântul-cheie *virtual*. De exemplu, următoarea clasă de bază *telephone* definește funcția membru *dial* ca fiind *virtuală*, astfel încât să poată lucra cu telefoane de tip *rotary* (cu disc rotativ) și *touch\_tone* (cu taste):

```
class telephone {
public:
    telephone(char *number, int volume);
    virtual int dial(char *outgoing_number);
protected:
    char phone_number[32];
    int volume;
};
```

Cuvântul - cheie *virtual*

Următoarele clase derivate, *rotary* și *touch\_tone*, sunt bazate pe clasa *telephone*:

```
class rotary: public telephone {
public:
    rotary(char *number, int volume) : telephone(number,
        volume) { };
    int dial(char *outgoing_number);
};
```

```
class touch_tone: public telephone {
public:
    touch_tone(char *number, int volume) : telephone(number,
        volume) { };
    int dial(char *outgoing_number);
};
```

După cum se observă, ambele clase definesc funcția membru *dial* (formează număr). Următorul program, TELEPHON.CPP, folosește aceste două clase și funcțiile *dial* aferente:

```
#include <iostream.h>
#include <string.h>
```

```
class telephone {
public:
    telephone(char *number, int volume);
    virtual int dial(char *outgoing_number);
protected:
    char phone_number[32];
    int volume;
};
```

```
telephone::telephone(char *number, int volume)
{
```

```
strcpy(telephone::phone_number, number);
telephone::volume = volume;
}

int telephone::dial(char *outgoing_number)
{
    cout << "Use a rotary or touch-tone phone to call: " <<
        outgoing_number;

    cout << " Volume: " << volume << endl;
    return(1);
}

class touch_tone: public telephone {
public:
    touch_tone(char *number, int volume) : telephone(number,
        volume) { };
    int dial(char *outgoing_number);
};

int touch_tone::dial(char *outgoing_number)
{
    cout << "Beep beep beep with touch tone: " << outgoing_number;
    cout << " Volume: " << volume << endl;

    return(1);
}

class rotary: public telephone {
public:
    rotary(char *number, int volume) : telephone(number,
        volume) { };
    int dial(char *outgoing_number);
};

int rotary::dial(char *outgoing_number)
{
    cout << "Click click click with rotary: " << outgoing_number;
    cout << " Volume: " << volume << endl;

    return(1);
}


void main(void)
{
    touch_tone office("363-1111", 5);
    rotary home("555-1234", 2);

    home.dial("222-3333");
    office.dial("333-4444");
}
```

Așa cum se observă, programul declară obiectele telefon *rotary* și *touch\_tone*, apoi le folosește pentru a face o legătură în vederea unei convorbiri. La compilarea și execuția programului, pe ecran va apare:

```
C:\> TELEPHON <ENTER>
Click click click with rotary: 222-3333 Volume: 2
Beep beep beep with touch tone: 333-4444 Volume: 5
```

În acest exemplu, programul folosește funcții virtuale, dar nu folosește polimorfismul. Cu alte cuvinte, obiectele folosite în program nu au mai multe forme - fiecare obiect este fie telefon cu disc (*rotary*) fie telefon cu taste (*touch\_tone*), dar nu de ambele tipuri.



### SĂ ÎNȚELEM FUNCȚIILE VIRTUALE

Polimorfismul din C++ se bazează pe *funcții virtuale*. În sensul cel mai simplu, o funcție virtuală este o funcție membru a unei clase care poate fi înlocuită, în momentul execuției programului, cu o funcție membru derivată. Este un fel de funcție "variabilă" pe care clasele derivate o pot substitui cu propriile lor funcții membru.

Pentru a declara o funcție virtuală, aceasta trebuie precedată de cuvântul-cheie *virtual*. De exemplu, clasa *computer* de mai jos folosește o funcție membru *processor* care este virtuală:

```
class computer {
public:
    computer(int processor_type);
    virtual processor(unsigned *instructions);
    void load_program(char *program_name);
private:
    int processor_type;
    int memory_size;
};
```

În acest fel, programul ar putea deriva, de exemplu, clasele PC și Mac, fiecare folosind o funcție membru *processor* specifică.

Programul anterior a folosit funcții virtuale, dar nu a realizat polimorfismul. Următorul program, POLY.CPP, modifică funcția *main* pentru a folosi un pointer la un obiect de tip *telephone* denumit *cellular*. În timpul execuției programului pointerul face referință atât la un obiect de tip *rotary*, cât și la unul de tip *touch\_tone*:

```
void main(void)
{
```

```
touch_tone office("363-1111", 5);
rotary home("555-1234", 2);
```

```
telephone *cellular;
```

```
home.dial("222-3333");
office.dial("333-4444");
```

```
cellular = &home;
cellular->dial("555-1234");
```

```
cellular = &office;
cellular->dial("555-5678");
}
```

**Observație:** Acest program nu conține și definițiile claselor și funcțiilor membru. După cum se observă, acest program atribuie variabilei pointer *cellular* adresele a două tipuri diferite de obiecte. La compilarea și execuția acestui program, ecranul va afișa următoarele:

```
Click click click with rotary: 222-3333 Volume: 2
Beep beep beep with touch tone: 333-4444 Volume: 5
Click click click with rotary: 555-1234 Volume: 2
Beep beep beep with touch tone: 555-5678 Volume: 5
```

Când pointerului *cellular* i se atribuie tipuri de obiecte diferite, programul va apela funcții distincte pentru funcția membru *dial*. Cu alte cuvinte, acest program realizează polimorfism, obiectul *cellular* luând diferite forme.

Exersați cu acest program, folosind un obiect de tipul *telephone*, în locul unui pointer la obiect. Următorul program, NOPOLY.CPP, atribuie obiectului *cellular* cele două obiecte derivate, *rotary* și *touch\_tone*. Veți vedea, totuși, că obiectul nu își schimbă forma, rămânând în permanență un obiect de tip *telephone*:

```
void main(void)
{
    touch_tone office("363-1111", 5);
    rotary home("555-1234", 2);

    telephone cellular("555-5555", 3);

    home.dial("222-3333");
    office.dial("333-4444");

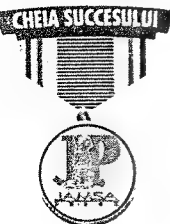
    cellular = home;
    cellular.dial("555-1234");
    cellular = office;
    cellular.dial("555-5678");
}
```

La compilarea și execuția programului, pe ecran va apărea:

```
C:\> NOPOLY <ENTER>
Click click click with rotary: 222-3333 Volume: 2
Beep beep beep with touch tone: 333-4444 Volume: 5
Use a rotary or touch-tone phone to call: 555-1234 Volume: 2
Use a rotary or touch-tone phone to call: 555-5678 Volume: 5
```

După cum se observă, când programul atribuie obiectului *cellular* cele două obiecte *rotary* și *touch\_tone*, forma obiectului *cellular* nu se schimbă. Cu alte cuvinte, obiectul *cellular* continuă să folosească funcția *dial* a clasei *telephone*.

**Observație:** Acest program nu conține și definițiile claselor și funcțiile membru.



### SĂ ÎNȚELEM POLIMORFISMUL

Polimorfismul este un alt mod de a spune „forme multiple”. Un obiect polimorfic este un obiect capabil de a-și schimba forma în timpul perioadei sale de existență. Polimorfismul este un instrument de programare puternic, deoarece permite scrierea unor fragmente de program ce se comportă corect pentru obiecte de tipuri diferite. În majoritatea cazurilor, tipul obiectului efectiv nu este cunoscut, nici chiar în timpul execuției. Așa cum veți învăța în Capitolul 13, polimorfismul se poate manifesta datorită unor operații efectuate de compilator „în spatele cortinei”. Deocamdată însă, să reținem că funcțiile virtuale C++ ne permit să scriem programe ce suportă polimorfismul, programe în care un obiect poate trece de la o formă la alta în timpul execuției.

### CE NU ESTE POLIMORFISMUL

Să nu confundăm polimorfismul cu suprapunerea funcțiilor. De exemplu, următorul program, *OVERONLY.CPP*, suprapune funcția *dial* în clasele *rotary* și *touch\_tone*. Deoarece definiția clasei de bază nu folosește cuvântul cheie *virtual*, funcțiile *dial* din clasele derivate vor supradefini funcția *dial* din clasa de bază. Funcțiile nu sunt, însă, virtuale:

```
#include <iostream.h>
#include <string.h>

class telephone {
public:
    telephone(char *number, int volume);
    int dial(char *outgoing_number);
protected:
    char phone_number[32];
```

Funcția nu este virtuală

```
int volume;
};

telephone::telephone(char *number, int volume)
{
    strcpy(telephone::phone_number, number);
    telephone::volume = volume;
}

int telephone::dial(char *outgoing_number)
{
    cout << "Use a rotary or touch-tone phone to call: " <<
        outgoing_number;

    cout << " Volume: " << volume << endl;
    return(1);
}

class touch_tone: public telephone {
public:
    touch_tone(char *number, int volume): telephone(number,
        volume) { };
    int dial(char *outgoing_number);
};

int touch_tone::dial(char *outgoing_number)
{
    cout << "Beep beep beep with touch tone: " << outgoing_number;
    cout << " Volume: " << volume << endl;

    return(1);
}

class rotary: public telephone {
public:
    rotary(char *number, int volume): telephone(number,
        volume) { };
    int dial(char *outgoing_number);
};

int rotary::dial(char *outgoing_number)
{
    cout << "Click click click with rotary: " << outgoing_number;
    cout << " Volume: " << volume << endl;

    return(1);
}

void main(void)
```

```
{
    touch_tone office("363-1111", 5);
    rotary home("555-1234", 2);

    telephone *cellular = new telephone("111-2222", 3);

    home.dial("222-3333");
    office.dial("333-4444");

    cellular->dial("123-4567");

    cellular = &office;

    cellular->dial("890-1234");
}
```

La compilarea și execuția programului, ecranul va afișa următoarele:

```
C:\> OVERONLY <ENTER>
Click click click with rotary: 222-3333 Volume: 2
Beep beep beep with touch tone: 333-4444 Volume: 5
Use a rotary or touch-tone phone to call: 123-4567 Volume: 3
Use a rotary or touch-tone phone to call: 890-1234 Volume: 5
```

Cu toate că obiectului *cellular* i s-a atribuit în program adresa obiectului de tip *office*, funcția membru *dial* nu s-a schimbat ca urmare a acestei atribuirii. Cu alte cuvinte, obiectul nu este polimorfic.

### FOLOSIREA FUNCȚIILOR MEMBRU ALE CLASEI DE BAZĂ

În exemplele anterioare, fiecare clasă derivată (*rotary* și *touch\_tone*) au definit funcția *dial* care a înlocuit funcția membru virtuală a clasei de bază.

În anumite cazuri, o clasă derivată nu va înlocui funcția virtuală, ci va folosi chiar funcția clasei de bază. Să presupunem de exemplu că scrieți un program de simulare pentru compania Rolls-Royce, care produce motoare de automobile, de camioane și chiar de avioane. Programul poate folosi clasa *engine*, ca mai jos:

```
class engine {
public:
    engine(char *name, int fuel_type);
    virtual void get_fuel(int gallons);
private:
    char name[64];
    int fuel_type;
};
```

Clasa *engine* definește funcția membru *get\_fuel* ca o funcție virtuală (fiecare motor folosește un combustibil diferit). În mod prestabilit, funcția *get\_fuel* a cla-

sei *engine* afișează un mesaj care înștiințează utilizatorul să încarce combustibil obișnuit sau fără plumb, cum se arată în continuare:

```
void engine::get_fuel(int gallons)
{
    cout << "Please add " << gallons << " of regular or unleaded" <<
        endl;
}
```

Pentru un motor de avion, însă, programul va cere utilizatorului să folosească un combustibil special de avion. De asemenea, pentru un camion, programul ar trebui să ceară un combustibil de tip Diesel. În sfârșit, pentru un motor de motocicletă, programul va folosi pur și simplu funcția prestabilită sau funcția clasei de bază. Următorul program, *ENGINE.CPP*, implementează clasa virtuală *engine*:

```
#include <iostream.h>
#include <string.h>

class engine {
public:
    engine(char *name, int fuel_type);
    virtual void get_fuel(int gallons);
private:
    char name[64];
    int fuel_type;
};

engine::engine(char *name, int fuel_type)
{
    strcpy(engine::name, name);
    engine::fuel_type = fuel_type;
}

void engine::get_fuel(int gallons)
{
    cout << "Please add " << gallons << " of regular or unleaded" <<
        endl;
}

class jet_engine : public engine {
public:
    jet_engine(char *name, int fuel_type) : engine(name,
        fuel_type) {};
    void get_fuel(int gallons)
    {
        cout << "Please add " << gallons << " of JP4 jet fuel" <<
            endl;
    }
};

class truck_engine : public engine {
```

```
public:
    truck_engine(char *name, int fuel_type) : engine(name,
        fuel_type) {};
    void get_fuel(int gallons)
    {
        cout << "Please add " << gallons << " of diesel fuel" <<
            endl;
    };
};
```

```
void main(void)
{
    engine rolls("Rolls Royce", 1);
    jet_engine F100("Jet aircraft", 2);
    truck_engine truck("Delivery truck", 3);
    engine harley("Harley Davidson", 1);

    rolls.get_fuel(20);
    F100.get_fuel(1000);
    truck.get_fuel(40);
    harley.get_fuel(5);
}
```

Se observă că obiectele *rolls* și *harley* folosesc funcția membru a clasei de bază. La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> ENGINE <ENTER>
Please add 20 of regular or unleaded
Please add 1000 of JP4 jet fuel
Please add 40 of diesel fuel
Please add 5 of regular or unleaded
```

Așa cum probabil ați observat, programul acesta nu ilustrează polimorfismul - nici un obiect nu își schimbă forma.

Următorul program, *USER\_ENG.CPP*, folosește obiectul pointer *user\_engine* pentru a afișa tipul de combustibil necesar unui anumit tip de motor, selectat de utilizator. Programul începe prin a cere utilizatorului să aleagă tipul de motor. În funcție de motorul selectat, pointerul *user\_engine* va lua una din formele posibile ale unui obiect de tip *engine*:

```
#include <ctype.h>

void main(void)
{
    engine rolls("Rolls Royce", 1);
    jet_engine F100("Jet aircraft", 2);
    truck_engine truck("Delivery truck", 3);
    engine harley("Harley Davidson", 1);
```

```
engine *user_engine;

int not_done = 1;

while (not_done)
{
    char user_choice;

    cout << "Select an engine type" << endl;
    cout << "S Standard\t Jet engine\tD Diesel\tQ Quit: " << endl;

    cin >> user_choice;

    user_choice = toupper(user_choice);

    switch (user_choice) {
        case 'S': user_engine = &rolls;
            break;
        case 'J': user_engine = &F100;
            break;
        case 'D': user_engine = &truck;
            break;
        case 'Q': not_done = 0;
            user_engine = (engine *) 0;
            break;
        default: user_engine = (engine *) 0;
            break;
    };

    if (user_engine)
    {
        user_engine->get_fuel(5);
        cout << endl << endl << endl;
    }
}
```

**Observație:** Acest program nu conține definițiile claselor și ale funcțiilor membru.

După cum se observă, programul atribuie un anumit tip de motor pointerului *user\_engine*, în funcție de răspunsul utilizatorului.

Deoarece pointerul *user\_engine* poate referi unul din mai multe tipuri de obiecte, el este un obiect polimorfic. Compilatorul nu poate determina în nici un fel la care obiect face pointerul referință la un anumit moment. Ca atare, polimorfismul apare la execuție, folosind suportul dat de compilatorul C++ ce va fi discutat în Capitolul 13.

## REGULI PENTRU FUNCȚII POLIMORFICE

Dacă nu precedați funcția membru a clasei de bază prin cuvântul-cheie *virtual*, polimorfismul nu va avea loc. În plus, dacă tipul rezultatului sau tipul parametrilor funcției din clasa derivată (cu același nume ca al funcției virtuale) nu sunt aceiași, polimorfismul nu va avea loc. De exemplu, următorul program, *MISMATCH.CPP*, modifică tipul parametrului *gallons* din *int* în *long*. Ca urmare, pointerul *engine* nu va fi polimorfic.

```
#include <iostream.h>
#include <string.h>

class engine {
public:
    engine(char *name, int fuel_type);
    virtual void get_fuel(int gallons);
private:
    char name[64];
    int fuel_type;
};

engine::engine(char *name, int fuel_type)
{
    strcpy(engine::name, name);
    engine::fuel_type = fuel_type;
}

void engine::get_fuel(int gallons)
{
    cout << "Please add " << gallons << " of regular or unleaded" << endl;
}

class jet_engine : public engine {
public:
    jet_engine(char *name, int fuel_type) : engine(name, fuel_type) {};
    void get_fuel(long gallons)
    {
        cout << "Please add " << gallons << " of JP4 jet fuel" << endl;
    }
};

void main(void)
{
    engine rolls("Rolls Royce", 1);
```

Definiția int a parametrului gallons

Definiția long a parametrului gallons

```
jet_engine F100("Jet aircraft", 2);

engine *engine_ptr;

engine_ptr = &rolls;
engine_ptr->get_fuel(20);

engine_ptr = &F100;
engine_ptr->get_fuel(1000);
}
```

Deoarece parametrii funcțiilor membru *get\_fuel* nu concordă în tip, polimorfismul nu apare. La compilarea și execuția programului, pe ecran va apărea:

```
C:\> MISMATCH <ENTER>
Please add 20 of regular or unleaded
Please add 1000 of regular or unleaded
```

Exersați cu acest program, modificând tipul *long* iarăși în *int*. Acum va apărea polimorfismul, iar programul va afișa:

```
C:\> MISMATCH <ENTER>
Please add 20 of regular or unleaded
Please add 1000 of JP4 jet fuel
```

## FUNCȚII VIRTUALE ȘI MOȘTENIREA PE MAI MULTE NIVELE

În Capitolul 2 ați examinat moștenirea claselor în detaliu. În funcție de structura claselor, apar situații când se folosește moștenirea pe mai multe nivele. De exemplu, să presupunem că vreți să creați o clasă *pay\_phone* (telefon public) bazată pe clasa *rotary* prezentată anterior:

```
class pay_phone: public rotary {
public:
    pay_phone(char *number, int volume, int cost): rotary(number, volume) {
        pay_phone::cost = cost;
    }
    int dial(char *outgoing_number);
private:
    int cost;
};
```

Să presupunem apoi că, clasa *pay\_phone* are nevoie de propria funcție *dial*, care cere utilizatorului să introducă o fisă de 100 lei. Se poate declara funcția membru *dial* ca virtuală în clasa *rotary* astfel:

```
class rotary: public telephone {
public:
```

```
rotary(char *number, int volume) : telephone(number,
volume) { };
virtual int dial(char *outgoing_number);
};
```

Declarând funcția ca virtuală, programul va putea crea obiecte cu propriile funcții membru *dial*, dar și obiecte ce folosesc funcția *dial* din clasa *rotary*. Următorul program, PAYPHONE.CPP, creează un obiect polimorfic ce poate fi fie *rotary* (telefon cu disc), fie de tipul *pay\_phone* (telefon public).

```
#include <iostream.h>
#include <string.h>

class telephone {
public:
    telephone(char *number, int volume);
    virtual int dial(char *outgoing_number);
protected:
    char phone_number[32];
    int volume;
};

telephone::telephone(char *number, int volume)
{
    strcpy(telephone::phone_number, number);
    telephone::volume = volume;
}

int telephone::dial(char *outgoing_number)
{
    cout << "Use a rotary or touch-tone phone to call: " <<
        outgoing_number;

    cout << " Volume: " << volume << endl;
    return(1);
}

class rotary: public telephone {
public:
    rotary(char *number, int volume) : telephone(number,
        volume) { };
    virtual int dial(char *outgoing_number);
};

int rotary::dial(char *outgoing_number)
{
    cout << "Click click click with rotary: " << outgoing_number;
    cout << " Volume: " << volume << endl;
    return(1);
}
```

```
class pay_phone: public rotary {
public:
    pay_phone(char *number, int volume, int cost): rotary(number,
        volume) {
        pay_phone::cost = cost;
    }
    int dial(char *outgoing_number);
private:
    int cost;
};

int pay_phone::dial(char *outgoing_number)
{
    cout << "Please deposit " << cost << " cents" << endl;

    return(rotary::dial(outgoing_number));
}

void main(void)
{
    pay_phone street("363-1111", 5, 25);
    rotary home("555-1234", 2);

    telephone *phone;

    phone = &home;
    phone->dial("222-3333");

    phone = &street;
    phone->dial("333-4444");
}
```

După cum se observă, pointerul *phone* face referință la două obiecte de tipuri diferite. La compilarea și execuția programului, pe ecran se va afișa:

```
C:\> PAYPHONE <ENTER>
Click click click with rotary: 222-3333 Volume: 2
Please deposit 25 cents
Click click click with rotary: 333-4444 Volume: 5
```

Se observă că clasa *pay\_phone* înlocuiește funcția membru *dial*. Examinați cu atenție cum clasa *pay\_phone* definește funcția *dial*:

```
int pay_phone::dial(char *outgoing_number)
{
    cout << "Please deposit " << cost << " cents" << endl;

    return(rotary::dial(outgoing_number));
}
```

La început, funcția înștiințează utilizatorul să introducă fisa corespunzătoare. Apoi, funcția folosește operatorul de rezoluție globală (::) pentru a apela funcția *dial* definită în clasa *rotary*, returnând valoarea returnată de *rotary::dial*. Experimentați acest program. Dacă, de exemplu, îndepărtați cuvântul cheie *virtual* din fața funcției *rotary::dial*, polimorfismul nu va mai avea loc.

## SĂ ÎNȚELEM FUNCȚIILE PUR VIRTUALE

*Abstractizarea* este procesul de ignorare temporară a detaliilor nesemnificative, în vederea concentrării asupra conceptelor importante ale unei probleme. În activitatea de programare, când se creează primele prototipuri ale unor programe mari, se inserează adesea definiții simple de funcții, ce vor fi ulterior înlocuite cu definițiile reale, mai complicate. De exemplu, un astfel de prototip ar putea fi:

```
void health_care_program(void)
{
    cout << "Health care program function-to be completed later" << endl;
}
```

Scriind astfel de funcții simple într-un program, se pot mai ușor depana și pune la punct anumite părți ale programului.

La proiectarea claselor unui program, se creează mai întâi clasele de bază, din care vor fi definite clasele ce vor fi utilizate în program. O *clasă abstractă* este o definiție de clasă al cărei scop este de a pune bazele pentru derivarea altor clase. În mod obișnuit, nu se creează obiecte dintr-o clasă abstractă. În schimb, clasele derivate vor moșteni caracteristicile clasei abstracte.

La folosirea claselor abstracte, apar situații când acestea conțin funcții virtuale ce pot fi folosite sau înlocuite de clasa derivată. În funcție de definiția clasei, există situații când se definește o funcție virtuală a cărei semnificație trebuie indicată de către clasa derivată. Funcțiile care trebuie definite de către clasa derivată se numesc *funcții pur virtuale*. Pentru a crea o funcție pur virtuală, aceasta trebuie inițializată cu zero în clasa abstractă. De exemplu, clasa *abstract\_telephone*, prezentată în continuare, definește funcția membru *dial* ca o funcție pur virtuală:

```
class abstract_telephone {
public:
    abstract_telephone(char *number, int volume);
    virtual int dial(char *outgoing_number) = 0; ———— Funcție pur virtuală
protected:
    char phone_number[32];
    int volume;
};
```

În cadrul clasei derivate, trebuie specificată o funcție pentru fiecare funcție pur virtuală. Următorul program, *VIRTDIAL.CPP*, folosește o funcție pur virtuală pentru *dial*:

```
#include <iostream.h>
#include <string.h>

class abstract_telephone {
public:
    abstract_telephone(char *number, int volume);
    virtual int dial(char *outgoing_number) = 0;
protected:
    char phone_number[32];
    int volume;
};

abstract_telephone::abstract_telephone(char *number, int volume)
{
    strcpy(abstract_telephone::phone_number, number);
    abstract_telephone::volume = volume;
}

class touch_tone: public abstract_telephone {
public:
    touch_tone(char *number, int volume) :
        abstract_telephone(number, volume) { };
    int dial(char *outgoing_number);
};

int touch_tone::dial(char *outgoing_number)
{
    cout << "Beep beep beep with touch tone: " << outgoing_number;
    cout << " Volume: " << volume << endl;

    return(1);
}

void main(void)
{
    touch_tone office("363-1111", 5);
    office.dial("333-4444");
}
```

La compilarea și execuția programului, pe ecran va apărea:

```
C:\> VIRTDIAL <ENTER>
Beep beep beep with touch tone: 333-4444 Volume: 5
```

Dacă continuați să exersați cu acest program și înălțurați funcția *touch\_tone::dial*, programul va fi compilat, dar programul de editare a legăturilor (linkeditare) va afișa mesaje de erori, care arată că funcția nu va fi găsită. Într-adevăr, clasele derivate trebuie să specifice definiții pentru funcțiile pur virtuale.





### SĂ ÎNȚELEGEM FUNCȚIILE PUR VIRTUALE

Dacă examinați definițiile claselor C++, veți putea întâlni funcții membru virtuale care sunt inițializate cu zero. Astfel de funcții, denumite funcții *pur virtuale*, sunt folosite în clasele abstracte pentru a specifica o funcție pe care clasele derivate din clasa abstractă trebuie să o definească ulterior. Dacă o clasă derivată nu definește o funcție pur virtuală, editorul de legături va afișa mesaje de eroare și nu va construi programul executabil.

### REZUMAT

Acest capitol a examinat polimorfismul și funcțiile virtuale în detaliu. Înainte de a trece la Capitolul 9, asigurați-vă că ați învățat următoarele:

- ✓ Polimorfismul este proprietatea unui obiect de a lua două sau mai multe forme diferite în timpul execuției programului. C++ suportă polimorfismul prin folosirea funcțiilor virtuale.
- ✓ O funcție virtuală este un fel de funcție "variabilă", o funcție membru executabilă, ce urmează a fi ulterior înlocuită cu funcții membru derivate. Este precedată de cuvântul-cheie virtual.
- ✓ Când într-un program se derivează o clasă dintr-o clasă de bază ce folosește o funcție virtuală, clasa derivată poate înlocui funcția virtuală cu o funcție proprie sau poate folosi chiar funcția definită în clasa de bază.
- ✓ Când o clasă derivată definește o funcție membru al cărui nume este același cu al funcției din clasa de bază, tipul rezultatului și tipul parametrilor trebuie să fie în concordanță cu tipul rezultatului și al parametrilor funcției din clasa de bază; în caz contrar polimorfismul nu va fi creat.
- ✓ O funcție pur virtuală este o funcție virtuală inițializată de clasa de bază cu 0. Clasa derivată trebuie să furnizeze o definiție pentru o funcție pur virtuală.

## CAPITOLUL 9 ACOMODAREA CU REGULILE DOMENIULUI DE VIZIBILITATE

În cadrul programelor, numele identifică variabile, funcții, clase, tipuri ș.a.m.d. *Domeniul* unui nume definește părțile programului în care numele are sens. De exemplu, variabilele cu *domeniu local* sunt vizibile în interiorul funcției în care sunt definite. În mod similar, variabilele cu *domeniu global* sunt vizibile în întreg programul, începând cu locul în care sunt declarate. Așa cum veți învăța în acest capitol, folosirea corectă a domeniului necesită mai mult decât cunoașterea domeniului local sau global al variabilelor. Când veți termina acest capitol, veți înțelege diferite aspecte ale conceptului de vizibilitate în C++. Exersați fiecare program în toate amănunțele. Multe din conceptele subtile ilustrate în programe pot fi sesizate numai după o examinare atentă. La încheierea capitolului, veți cunoaște:

- ♦ Ce este domeniul unui nume (identificator)
- ♦ Ce sunt domeniul local și domeniul global
- ♦ Cum se indică scopul unui identificator
- ♦ Care sunt cele 4 tipuri de domeniu
- ♦ Ce este domeniul unei funcții
- ♦ Ce este domeniul unui fișier
- ♦ Ce este domeniul unei clase
- ♦ Unde se pot declara variabilele într-un program C++
- ♦ Ce se întâmplă când numele unei variabile locale intră în conflict cu numele unei variabile globale sau al unei variabile de clasă
- ♦ Ce este durata de viață a unei variabile
- ♦ Ce este legarea internă și externă

### DECLARAȚII ȘI DEFINIȚII

Când citim cărți sau articole despre C sau C++, termenii *definiție* și *declarație* se folosesc deseori alternativ. Cu toate că par a fi sinonime, sensurile celor două cuvinte diferă. O declarație specifică proprietățile unei variabile: un tip (ca *int*,

*float* sau *char*) și un nume. De exemplu, următoarele instrucțiuni sunt declarații valide în C sau C++:

```
int age, count;
float salary;
long int miles_to_the_moon;
```

```
struct date {
    int day;
    int month;
    int year } birthday;
```

```
enum suite { clubs, diamonds, hearts, spades };
```

În general, când o declarație are și efectul alocării de memorie pentru stocarea obiectului, atunci această declarație este și o *definiție*. Fiecare din declarațiile anterioare au alocat și memorie pentru entitățile declarate, deci sunt și definiții.

Următoarele declarații, însă, nu alocă memorie, deci nu sunt definiții:

```
extern int error_flag;
extern istream_withassign _Cdecl cin;
extern ostream_withassign _Cdecl cout;
```

```
int some_function(int, float, char); // Function prototype
```

În acest caz, declarațiile specifică tipul și numele fiecărei variabile. Totuși, cuvântul cheie *extern* informează compilatorul că variabilele sunt *definite* în altă parte. Prin urmare, compilatorul nu va alocă memorie pentru stocarea variabilelor. Așa cum arată prototipul funcției anterioare, declarațiile nu sunt caracteristice numai variabilelor. Prototipurile de funcții sunt de asemenea declarații, deoarece introduc în program numele funcției. Instrucțiunile funcției, pe de altă parte, definesc funcția.

În sensul cel mai simplu, o declarație introduce un nume. Pentru compilatorul C++, fiecare nume are asociat un tip, care determină setul de valori pe care îl poate lua entitatea respectivă și setul de operații ce poate fi realizat pe acea entitate. Domeniul unui nume specifică zona programului unde numele are sens. Datorită domeniului de vizibilitate, același nume poate fi folosit în mod diferit, în diferite părți ale programului.

## PRINCIPII DE BAZĂ ALE DOMENIULUI DE VIZIBILITATE

Fiecare entitate (variabilă, funcție, clasă etc.) dintr-un program are un nume. Domeniul numelui entității definește acea zonă a programului unde numele are sens. C++ definește 4 tipuri de domeniu: domeniul local, domeniul funcției, domeniul fișierului, domeniul clasei.



```
int i, j, k;
float salary;
char string[256];
extern istream_withassign _Cdecl cin;
```

### SĂ ÎNȚELEGEM DECLARAȚIILE ȘI DEFINIȚIILE

În cadrul unui program, variabilele, clasele, funcțiile și toate celelalte entități au nume. Înainte de a referi (utiliza) o entitate, numele ei trebuie să fie mai întâi declarat. O declarație specifică un nume și un tip, similar celor arătate mai jos:

În sensul cel mai simplu, dacă o declarație alocă memorie pentru o anumită entitate, devine automat și o definiție. În cazul declarațiilor anterioare, numai ultima instrucțiune (care folosește cuvântul cheie *extern*) nu alocă memorie. Prin urmare, acea instrucțiune nu este o definiție. Mai exact, totuși, o declarație este o definiție numai dacă nu intră într-una din următoarele categorii:

- declară prototipul unei funcții
- conține cuvântul cheie *extern*, dar nu face inițializări și nu conține instrucțiuni din structura unei funcții
- declară numele unei clase
- declară un membru de clasă statică
- declară un nou tip prin *typedef*

### SĂ ÎNȚELEGEM DOMENIUL LOCAL

*Domeniul local* corespunde acelor entități definite într-un bloc al programului (între acolade), ori parametrilor formali dintr-o definiție de funcții. Variabilele cu domeniu local sunt cunoscute numai în interiorul blocului în care sunt declarate, începând cu locul declarației. De exemplu, următorul program, LOCAL.CPP, creează 3 variabile locale, denumite *chapter*, *title* și *book*:

```
#include <iostream.h>
```

```
void main(void)
{
```

```
    int chapter = 9;
    char *title = "Getting Up to Speed with Rules of Scope";
    char *book = "Success with C++";
```

Declarații variabile locale

```
cout << "Chapter " << chapter << " " << title << endl;
cout << book << endl;
}
```

În C, variabilele locale pot fi definite la începutul oricărui bloc al programului. În C++, însă, variabilele pot fi definite în orice poziție, chiar combinate cu instrucțiunile programului. Prin urmare, următorul program, MIXLOCAL.CPP, este identic din punct de vedere funcțional cu programul anterior:

```
#include <iostream.h>

void main(void)
{
    int chapter = 9;
    char *title = "Getting Up to Speed with Rules of Scope";

    cout << "Chapter " << chapter << " " << title << endl;

    char *book = "Success with C++";
    cout << book << endl;
}
```

Prin declararea variabilelor în apropierea locului de folosire, se poate îmbunătăți claritatea programelor mari. Un motiv pentru a declara variabilele în interiorul instrucțiunilor programului este de a putea declara variabila de control în interiorul unui ciclu *for*, ca mai jos:

Declarație variabilă locală

```
for (int i = 0; i < 50; i++)
    some_array[i] = 0;
```

Se observă că ciclul *for* declară și inițializează variabila *i* în interiorul ciclului. După încheierea ciclului, variabila locală *i* rămâne definită. Următorul program, SHOW\_I.CPP, definește o variabilă *i* în interiorul ciclului *for*. După terminarea ciclului, programul afișează valoarea variabilei:

```
#include <iostream.h>

void main(void)
{
    for (int i = 0; i < 25; i++)
        cout << i;

    cout << endl << "Value of i is " << i << endl;
}
```



### DECLARAREA VARIABILELOR ÎN INTERIORUL UNUI BLOC DE INSTRUCȚIUNI

C++ permite programelor să declare variabilele și între instrucțiuni, nu numai la începutul blocurilor de instrucțiuni. În multe situații, se poate îmbunătăți claritatea unui program dacă variabilele se declară în apropierea locului lor de folosire. De exemplu, următoarele instrucțiuni declară variabilele *age* și *name* între instrucțiunile programului:

```
cout << "Type in the employee name and age: ";

int age;
char name[64];

cin >> name >> age;
```

În mod similar, următoarea instrucțiune declară variabila *i* într-o instrucțiune *for*:

```
for (int i = 0; i < 10; i++)
    cout << i << endl;
```

Modul de declarare a variabilelor într-un program este, ca și alte chestiuni de ingineria programelor, discutabil. Ca o regulă, totuși, dacă declararea unei variabile între instrucțiunile programului îmbunătățește claritatea acestuia, atunci este recomandabilă.

Așa cum s-a arătat, numele variabilelor locale au sens numai în interiorul blocului în care sunt definite. Dacă programul încearcă să folosească o variabilă într-un bloc înainte de declararea ei, sau în alt bloc disjunct, numele variabilei este necunoscut și va apărea o eroare de sintaxă. Următorul program, TWOBOOKS.CPP, de exemplu, creează o variabilă locală denumită *book* și atribuie acesteia un șir de caractere. În interiorul unei instrucțiuni *if*, programul creează o a doua variabilă locală denumită *book*, căreia i se atribuie altă valoare. Deoarece fiecare variabilă are propriul ei domeniu de vizibilitate, cele două variabile vor fi distincte, chiar dacă au același nume:

```
#include <iostream.h>

void main(void)
{
    char *book = "Success with C++"; // Prima declarație a variabilei book

    if (book)
```

```
{
    char *book = "Rescued by C++"; // A doua declarație a variabilei book
    cout << "Start with the book " << book << endl;
}


cout << "Then read the book " << book << endl;
}
```

**Observație:** Acest program este prezentat pentru a ilustra domeniul local, nefiind un exemplu bun de programare. De regulă, e bine ca funcțiile să nu aibă două sau mai multe variabile cu același nume. Folosirea aceluiași nume pentru variabile diferite face ca programul să fie greu de înțeles.

Câteva definiții prezentate în acest capitol folosesc expresia „entitatea poate fi folosită oriunde în blocul în care a fost declarată, începând cu locul declarării”. Aceasta înseamnă că, dacă o variabilă este declarată în mijlocul unui bloc, ea nu va putea fi referită anterior declarației. Variabila va fi cunoscută (în interiorul blocului respectiv) numai după declarare. Următoarele instrucțiuni vor genera, prin urmare, o eroare de sintaxă:

```
cout << "This is chapter " << chapter;
int chapter = 9;
```

Eroarea apare deoarece programul încearcă să folosească variabila *chapter* înainte de a fi fost declarată.



**SĂ ÎNȚELEM DOMENIUL LOCAL**

Variabilele cu domeniu local sunt cunoscute numai în blocul în care sunt declarate, începând cu locul declarației. C++ permite programelor să declare variabile locale în orice parte a programului, chiar intercalate cu instrucțiunile programului. Declarând variabilele aproape de locul folosirii lor, se poate îmbunătăți claritatea programelor. C++ consideră parametrii formali ai funcțiilor ca variabile locale, care sunt cunoscute numai în interiorul funcției.

### SĂ ÎNȚELEM DOMENIUL UNEI FUNCȚII

O etichetă definește o locație situată între instrucțiunile unei funcții. Folosind instrucțiunea *goto*, programul poate transfera controlul instrucțiunii ce urmează după etichetă. Următorul program, GOTOLOOP.CPP, folosește instrucțiunea *goto* și eticheta *loop* pentru a afișa pe ecran numerele naturale între 1 și 50:

```
#include <iostream.h>


void main(void)
```

```
{
    int i = 1;

    loop:
        cout << i++ << endl;

    if (i <= 50)
        goto loop;
}
```

La folosirea instrucțiunii *goto*, eticheta la care se face transferul trebuie să se afle în interiorul aceleiași funcții ca și *goto*. Cu alte cuvinte, etichetele au domeniu de funcție. Numele etichetelor sunt singurele entități care au un astfel de domeniu.



**SĂ ÎNȚELEM DOMENIUL UNEI FUNCȚII**

Domeniul unei funcții restricționează valabilitatea unui identificador în interiorul funcției în care este declarat. Numele de etichete sunt singurele entități în C++ care au domeniu de funcție. Prin urmare, instrucțiunea *goto* nu poate efectua transferul de la o funcție la o etichetă definită în altă funcție.

### SĂ ÎNȚELEM DOMENIUL UNUI FIȘIER

Variabilele declarate într-un fișier sursă, dar în afara programului, sunt variabile cu domeniu fișier. În sensul cel mai simplu, variabilele cu domeniu fișier sunt variabile globale. Variabilele globale sunt recunoscute în toate blocurile programului care urmează după declarație. Următorul program, GLOBAL.CPP, declară variabilele *book* și *chapter* ca variabile globale. Programul afișează apoi valoarea fiecărei variabile în două funcții diferite.

```
#include <iostream.h>

char *book = "Success with C++";
char *chapter = "Getting Up to Speed with Scope";

void show_globals(int count)
{
    for (int i = 0; i < count; i++)
        cout << book << " " << chapter << endl;
}


void main(void)
{
    cout << "This book: " << book << endl;
    cout << "Chapter 13: " << chapter << endl;
}
```

```
show_globals(1);
}
```

Reamintim că programele nu pot utiliza o variabilă globală înainte de a fi declarată. Următoarele instrucțiuni, de exemplu, încearcă să folosească variabila globală *error\_member* în funcția *show\_error*, înainte ca variabila să fie declarată. La compilarea acestor instrucțiuni va apărea o eroare de sintaxă:

```
void show_error(void)
{
    cerr << "Error in processing # " << error_number << endl;
}

int error_number; // Global variable
```



### SĂ ÎNȚELEGEM DOMENIUL UNUI FIȘIER

Domeniul unui fișier specifică faptul că variabilele declarate în afara tuturor blocurilor programului sunt recunoscute în toate blocurile care urmează după declarația variabilelor. Variabilele având domeniul fișier sunt globale. Deoarece variabilele globale sunt ușor accesibile în întregul program, ele sunt foarte susceptibile la erori. De aceea, de regulă, programele trebuie să restrângă la maximum folosirea variabilelor globale.

### SĂ ÎNȚELEGEM DOMENIUL UNEI CLASE

Clasele asigură camuflajul datelor și încapsularea. În interiorul unei clase, membrii acesteia sunt locali și pot fi folosiți numai de funcțiile membru ale clasei (sau de funcțiile din clasele derivate), cu operatorul punct (.) sau de indirectare (→), după operatorul de rezoluție (::), sau după clasele declarate ca *friend*. Această secțiune examinează diferite operații cu membrii clasei, care dispun de domeniu clasă.

### SĂ ÎNȚELEGEM MEMBRII DE TIP PUBLIC ȘI PRIVATE AI UNEI CLASE

Așa cum ați citit în Capitolul 2, membrii unei clase pot fi de tip *public* sau *private*, în funcție de modul în care dorim ca programul să aibă acces la variabile. Membrii de tip *private* pot fi folosiți numai de funcțiile membru ale clasei (sau funcțiile claselor derivate sau funcțiile *friend*). Membrii de tip *public*, pe de altă parte, pot fi apelați din orice parte a programului folosind operatorul punct (.) sau de indirectare (→). De exemplu, următorul program, PUBPRIV.CPP, crează o clasă *Book*, care conține variabilele membru *title*, *author* și *pages*. Clasa conține, de asemenea, funcția constructor *Book* și funcția membru *show\_book*:

```
class Book {
public:
```

```
Book(char *book_title, char *book_author, int book_pages)
{
    strcpy(title, book_title);
    strcpy(author, book_author);
    pages = book_pages;
};

void show_book(void)
{
    cout << "Book: " << title << endl;
    cout << "Author: " << author << endl;
    cout << "Pages: " << pages << endl;
}

private:
    char title[64];
    char author[64];
    int pages;
};
```

După cum se vede, funcțiile membru ale clasei sunt implementate folosind instrucțiuni inline. Definind variabilele membru de tip *private*, programul restricționează accesul la acestea numai la funcțiile membru ale clasei. Următoarele instrucțiuni implementează programul PUBPRIV.CPP:

```
#include <iostream.h>
#include <string.h>


class Book {
public:
    Book(char *book_title, char *book_author, int book_pages)
    {
        strcpy(title, book_title);
        strcpy(author, book_author);
        pages = book_pages;
    };

    void show_book(void)
    {
        cout << "Book: " << title << endl;
        cout << "Author: " << author << endl;
        cout << "Pages: " << pages << endl;
    }

private:
    char title[64];
    char author[64];
    int pages;
};


void main(void)
{
    Book cpp_book("Success with C++", "Jamsa", 528);
    cpp_book.show_book();
}
```

Așa cum se observă, clasa *Book* nu permite accesul programului la variabilele membru, definindu-le de tip *private*. Singurul mod de a avea acces la variabilele membru este prin intermediul funcțiilor membru ale clasei de tip *public*, cum sunt funcția constructor sau funcția *show\_book*.



**FOLOSIREA MEMBRILOR DE TIP PUBLIC ȘI PRIVATE AI UNEI CLASE**

Când creai propriile clase, cuvintele cheie *private* și *public* permit programului să controleze accesul la membri, asemănător domeniului variabilelor. De regulă, se declară variabilele membru de tip *private* cât mai des posibil, permițând accesul la variabile numai prin funcțiile membru. De asemenea, numai acele funcții pe care programul le folosește direct trebuie să fie de tip *public*. Cu alte cuvinte, folosiți teoria „trebuie să știu” pentru a determina care membri trebuie să fie de tip *public* și care de tip *private*.



**SĂ ÎNȚELEM ACCESUL CONTROLAT**

Când citiți despre avantajele de a defini variabilele clasei de tip *private* întâlniți și sintagma *acces controlat*. Când variabilele membru sunt de tip *public*, programul poate schimba variabilele oricând și oricum. Să presupunem, totuși, că ați creat o clasă ce modelează un reactor nuclear. Una dintre variabilele membru e denumită *protecție\_la\_scurgere*. Când variabila conține valoarea 1, reactorul nuclear este protejat împotriva scurgerilor. Când variabila conține valoarea 0, se presupune că reactorul e decuplat și că protecția poate fi dezactivată. Dacă variabila membru este de tip *public*, un programator ar putea intenționa să schimbe valoarea variabilei folosind o instrucțiune de tipul:

```
reactor.protecție_la_scurgere = 0;
```

De asemenea, programatorul ar putea accidental să schimbe valoarea variabilei printr-o eroare de programare. De exemplu, următoarea instrucțiune *if* se presupune că verifică dacă valoarea variabilei este 0. Din păcate, deoarece programatorul a folosit operatorul de atribuire (=) în locul testului la egalitate (==), variabilei membru i se atribuie valoarea 0:

```
if (reactor.protecție_la_scurgere = 0)
```

Pentru a evita astfel de erori, variabila ar trebui folosită numai prin intermediul funcțiilor membru, ca *setează\_protecția*, care poate cere utilizatorului o parolă, sau *obține\_protecția*, care returnează valoarea curentă a variabilei. Acesta este *accesul controlat*. Când scrieți un program, cunoașteți cum trebuie folosită fiecare variabilă membru. Declarând acele variabile de tip *private*, vă puteți asigura că ele vor fi folosite așa cum ați dorit.

Cu domeniul clasei, variabilele membru ale clasei pot fi direct apelate de funcțiile membru. De exemplu, să considerăm fie constructorul *Book*, fie funcția membru *show\_book* prezentate anterior:

```
Book(char *book_title, char *book_author, int book_pages)
{
    strcpy(title, book_title);
    strcpy(author, book_author);
    pages = book_pages;
};

void show_book(void)
{
    cout << "Book: " << title << endl;
    cout << "Author: " << author << endl;
    cout << "Pages: " << pages << endl;
}
```

Se observă că ambele funcții referă direct variabilele membru prin nume. În cazul funcției constructor *Book*, aceasta atribuie funcțiilor membru valorile parametrilor. Când funcțiile membru primesc parametri, sunt situații când numele unui parametru poate coincide cu acela al unei variabile membru. De exemplu, să presupunem că funcția constructor *Book* a folosit numele de parametri *title*, *author* și *price*, ca mai jos:

```
Book(char *title, char *author, int pages)
```

Când numele variabilelor locale (rețineți că parametrii formali sunt variabile locale) sunt aceleași cu ale membrilor clasei, membrii clasei devin *ascunși*. Operațiile din corpul funcției care referă numele variabilei vor fi aplicate variabilei locale, nu variabilei membru. În astfel de situații, funcția poate referi direct funcțiile membru folosind operatorul de rezoluție globală (::), ca mai jos:

```
Book(char *title, char *author, int pages)
{
    strcpy(Book::title, title);
    strcpy(Book::author, author);
    Book::pages = pages;
};
```

Următorul program, UNHIDE.CPP, folosește operatorul de rezoluție globală pentru a folosi variabile membru ce au fost ascunse de variabilele locale:

```
#include <iostream.h>
#include <string.h>

class Book {
public:
    Book(char *title, char *author, int pages)
```

```

{
    strcpy(Book::title, title);
    strcpy(Book::author, author);
    Book::pages = pages;
};

void show_book(void)
{
    cout << "Book: " << title << endl;
    cout << "Author: " << author << endl;
    cout << "Pages: " << pages << endl;
}

private:
char title[64];
char author[64];
int pages;
};

void main(void)
{
    Book cpp_book("Success with C++", "Jamsa", 528);

    cpp_book.show_book();
}

```

În cazul programului precedent, clasa *Book* a folosit funcții inline. Așa cum ați citit în Capitolul 2, când o clasă folosește funcții inline, fiecare obiect primește câte un exemplar din codul funcției. Cu alte cuvinte, programul nu beneficiază de partajarea instrucțiunilor de către mai multe obiecte. Următorul program, *NOINLINE.CPP*, deplasează funcțiile clasei în afara definiției acesteia:

```

#include <iostream.h>
#include <string.h>

class Book {
public:
    Book(char *title, char *author, int pages);
    void show_book(void);
private:
    char title[64];
    char author[64];
    int pages;
};

Book::Book(char *title, char *author, int pages)
{
    strcpy(Book::title, title);
    strcpy(Book::author, author);
    Book::pages = pages;
};

```

Funcția constructor  
a clasei *Book*

```

void Book::show_book(void)
{
    cout << "Book: " << title << endl;
    cout << "Author: " << author << endl;
    cout << "Pages: " << pages << endl;
}

void main(void)
{
    Book cpp_book("Success with C++", "Jamsa", 528);

    cpp_book.show_book();
}

```

Funcție membru a  
clasei *Book*

După cum se observă, programul folosește operatorul de rezoluție globală cu numele clasei *Book* pentru a avea acces la funcțiile membru ale clasei. Pe măsură ce obiectele dintr-un program devin mai complexe, poate apărea o coincidență de nume între funcțiile membru ale unei clase și funcțiile membru ale altei clase. Aceste conflicte de nume se pot rezolva folosind operatorul de rezoluție globală precedat de numele clasei.



#### ATENȚIE LA MEMBRII ASCUNȘI

La transferul parametrilor către funcțiile membru ale clasei, este posibil ca numele parametrilor (variabilelor locale) să coincidă cu cel al membrilor clasei. Când apar astfel de conflicte, C++ ascunde membrul clasei, asociind toate referințele către numele de conflict la variabila locală (parametru). Pentru a putea referi, în astfel de situații, membrul clasei, trebuie folosit operatorul de rezoluție globală (::). De exemplu, următoarea funcție, *set\_salary*, atribuie valoarea specificată în parametrul *salary* variabilei membru *salary* din clasa *employee*:

```

void employee::set_salary(float salary)
{
    employee::salary = salary;
}

```

Așa cum se observă, funcția folosește operatorul de rezoluție globală cu numele clasei pentru a putea avea acces la variabila membru ascunsă.

#### SĂ ÎNȚELEM CLASELE FRIEND

Așa cum ați învățat, membrii de tip *private* ai unei clase permit un acces controlat asupra clasei respective. Când construți programe cu obiecte dependente, claritatea programului sau camuflajul informației se pot îmbunătăți prin acordarea drepturilor de acces la unul sau mai mulți membri de tip *private* ai clasei

unei anumite clase, denumită *friend* (prietenă). Să presupunem, de exemplu, că un program folosește clasa *employee*, care conține membri de tip *public* și *private*, cum se arată mai jos:

```
class employee {
public:
    employee(char *name, float salary, char *ssan);
    void show_employee(void);
    void change_employee(void);
private:
    char name[256];
    float salary;
    char ssan[256];
};
```

Apoi, să presupunem că în program există și clasa *manager* care trebuie să aibă acces direct la membrii de tip *private* ai clasei *employee*. Una din soluții ar fi să declarăm de tip *public* toți membrii clasei *employee*, pentru a-i face accesibili funcțiilor din clasa *manager*. Dar, din păcate, aceștia ar deveni accesibili în întregul program. O soluție bună ar fi ca membrii să fie accesibili numai clasei *manager*, specificând clasa *manager* ca fiind *friend* al clasei *employee*:

```
class employee {
public:
    employee(char *name, float salary, char *ssan);
    void show_employee(void);
    void change_employee(void);
    friend manager; // Specificare drept friend a obiectelor
                    // din clasa manager
private:
    char name[256];
    float salary;
    char ssan[256];
};
```

Următorul program, MGR\_UPD.CPP, creează obiectele *employee* și *manager*. Programul transmite apoi obiectul *employee* funcției membru *update\_employee* a clasei *manager*, care actualizează obiectul. Pentru a permite funcției să schimbe obiectul *employee*, programul transmite funcției obiectul prin adresă:

```
#include <iostream.h>
#include <string.h>

class manager;

class employee {
public:
    employee(char *name, float salary, char *ssan);
    void show_employee(void);
    void change_employee(char *name, float salary, char *ssan);
    friend manager;
private:
```

```
char name[256];
float salary;
char ssan[256];
};

employee::employee(char *name, float salary, char *ssan)
{
    strcpy(employee::name, name);
    strcpy(employee::ssan, ssan);
    employee::salary = salary;
}

void employee::show_employee(void)
{
    cout << "Name: " << name << endl;
    cout << "Salary: " << salary << endl;
    cout << "SSAN: " << ssan << endl;
}

void employee::change_employee(char *name, float salary, char *ssan)
{
    strcpy(employee::name, name);
    strcpy(employee::ssan, ssan);
    employee::salary = salary;
}

class manager {
public:
    void update_employee(employee *emp, char *name, float salary,
        char *ssan);
    // Other members
};

void manager::update_employee(employee *emp, char *name,
    float salary, char *ssan)
{
    emp->change_employee(name, salary, ssan);
}

void main(void)
{
    employee worker("Jamsa", 25000.0, "538-66-5444");

    manager the boss;

    worker.show_employee();

    the boss.update_employee(&worker, "Jones", 33456.0,
        "111-22-3333");

    worker.show_employee();
}
```



Să observăm declarația clasei *manager* la începutul programului:

```
class manager;
```

Programul folosește această declarație pentru a permite obiectului *employee* să refere clasa drept *friend*. Așa cum se vede, programul transmite obiectul *employee* funcției *update\_employee* prin adresă, ceea ce permite programului să modifice variabilele membru ale obiectului. Exersați cu acest program și îndepărtați instrucțiunea *friend* din declarația clasei *employee*. Vor apărea erori de sintaxă, deoarece accesul obiectului *manager* la funcția *change\_employee* nu va mai fi permis.

Declarând clasa *manager* ca *friend*, programul anterior a oferit obiectelor *manager* acces deplin la membrii obiectului *employee*. Din păcate, declarând clasa *manager* ca *friend*, obiectele de tip *manager* pot avea acces complet la membrii obiectului *employee*, în orice mod. În multe cazuri este de dorit să reducem accesul unei clase *friend* la anumite funcții. De exemplu, următoarea declarație de clasă *employee* acordă drepturi de acces numai funcției *update\_employee* a clasei *manager*:

```
class employee {
public:
    employee(char *name, float salary, char *ssan);
    void show_employee(void);
    void change_employee(char *name, float salary, char *ssan);
    friend void manager::update_employee(employee *emp, char *name,
        float salary, char *ssan);
private:
    char name[256];
    float salary;
    char ssan[256];
};
```

Restrângând accesul în acest mod se poate controla mai exact accesul programului la obiecte, facilitând înțelegerea programului și reducând șansele de erori.



#### ACCESUL SPECIAL AL CLASELOR FRIEND LA MEMBRII UNUI OBIECT

Dacă numărul de obiecte într-un program crește, se poate atunci îmbunătăți performanța sau claritatea programului printr-o clasă de obiecte ce au acces special la membrii unui obiect. Dacă într-o declarație de clasă declarăm o altă clasă ca fiind prieten (*friend*) al obiectului, atunci oferim clasei

*friend* accesul complet la membrii de tip *private* ai obiectului. Următoarea clasă, *comedian*, de exemplu, declară obiectele clasei *agent* ca fiind de tip *friend*. În acest fel, obiectele *agent* au acces la membrii de tip *private* ai clasei *comedian*.

```
class agent;
class comedian {
public:
    comedian(char *name, char *joke);
    void show_comedian(void);
    friend agent;
private:
    char *name[64];
    char *best_joke[256];
};
```

De observat declarația simplă a clasei *agent* ce precede declarația clasei *comedian*. În felul acesta, clasa *comedian* poate referi clasa *agent* înainte ca aceasta din urmă să fie declarată formal. În unele cazuri este necesară reducerea accesului clasei *friend* numai la una dintre funcțiile sale membru. Pentru aceasta, se specifică funcția în cadrul declarației clasei, ca mai jos:

```
class comedian {
public:
    comedian(char *name, char *joke);
    void show_comedian(void);
    friend void agent::pay_comic(void);
private:
    char *name[64];
    char *best_joke[256];
};
```

În acest caz numai funcția membru *pay\_comic* a clasei *agent* are acces la variabilele membru de tip *private* ale clasei *comedian*.

#### SĂ ÎNȚELEM MEMBRII DE TIP PROTECTED

În Capitolul 4 s-a discutat moștenirea în detaliu. Așa cum ați învățat, la derivarea unei clase dintr-o clasă de bază, clasa derivată va avea acces la membrii de tip *public* ai clasei de bază, dar nu și la membrii de tip *private*. Pentru a oferi claselor derivate un acces de nivel mediu, C++ admite *membri protejați* (*protected members*) ai unei clase. Membrii protejați pot fi apelați de clasele derivate, dar nu și de restul programului. Următorul program, PROTECT.CPP, derivează clasa *document* folosind clasa de bază *file*:

```
#include <iostream.h>
#include <string.h>

class file {
public:
    file(char *name, int type);
protected:
    void show_file_info(void);
    int attributes;
private:
    ...
```

```

char name[64];
int type;
};

file::file(char *name, int type)
{
    strcpy(file::name, name);
    file::type = type;
}

void file::show_file_info(void)
{
    cout << "Filename: " << name << endl;
    cout << "Type: " << type << endl;
}

class document : public file {
public:
    document(char *name, int type, int attributes):
        file(name, type)
    {
        document::attributes = attributes;
    }

    void document::show_document(void)
    {
        show_file_info();
        cout << "Attributes: " << attributes << endl;
    }

void main(void)
{
    document spreadsheet("WORKSHEET.DAT", 1, 255);

    spreadsheet.show_document();
}

```

După cum se observă, clasa *file* declară membrii *attributes* și *show\_file\_info* de tip *protected*. Exersați cu acest program, încercând să apelați cei doi membri direct din funcția *main*. Veți vedea că singurul mod de a avea acces la membrii protejați este prin funcțiile membru ale claselor *file* și *document*.



#### FOLOSIREA MEMBRILOR DE TIP PROTECTED

La folosirea conceptului de moștenire din C++ pentru derivarea unei clase dintr-o altă clasă de bază, este posibil ca unii membri ai clasei de bază să poată fi utilizați de obiectele din clasa derivată, dar nu și din restul programului. În astfel de situații, se pot folosi membrii de tip *protected*. Aceștia sunt accesibili pentru funcțiile membru ale clasei de bază sau ale clasei derivate, dar nu și din afara acestor funcții.

#### EFFECTUL CALIFICATORILOR ASUPRA DOMENIULUI

Așa cum știți, la declararea variabilelor, C++ permite adăugarea unor *calificatori* la declararea tipurilor, ca *long*, *short* și alții. În plus, puteți preceda declararea variabilelor cu *specificatorii clasei de memorie*, ca *static*, *auto*, *extern* sau *register*. Această secțiune examinează specificatorii *extern* și *static* și influența lor asupra domeniului variabilelor, funcțiilor și al membrilor clasei. Veți vedea că, folosind acești calificatori, se poate controla accesul la diferiți identificatori.

#### FOLOSIREA CALIFICATORULUI EXTERN

Așa cum ați învățat, diferența între declarații și definiții rezidă în alocarea spațiului de memorie. Dacă precedați o declarație de cuvântul cheie *extern*, înștiințați compilatorul C++ că folosiți un identificator care a fost definit în altă parte (în fișierul sursă al programului, în fișierul antet sau într-un fișier de bibliotecă). Următorul program, EXTERN.CPP, folosește variabila *title*, care este definită în afara programului:

```

#include <iostream.h>

void show_title(void);

void main(void)
{
    extern char *title;

    cout << "The title is " << title << endl;

    show_title();
}

```

După cum se vede, programul declară variabila *title* ca de tip *extern*. În acest fel, compilatorul C++ nu alocă spațiu pentru variabilă. De fapt, variabila *title* este definită în fișierul sursă SHOWTITL.CPP, care definește și funcția *show\_title*:

```
#include <iostream.h>

char *title = "Success with C++";

void show_title(void)
{
    cout << "The title of this book is " << title << endl;
}
```

În cazul că folosiți compilatorul Borland C++, programul se poate compila astfel:

```
C:\> BCC EXTERN.CPP SHOWTITL.CPP <ENTER>
```

La execuția programului, pe ecran va apărea:

```
C:\> EXTERN <ENTER>
The title is Success with C++
The title of this book is Success with C++
```

Exersați cu acest program, înlăturând eventual cuvântul *extern* din fața declarației variabilei *title*, sau mutând declarația acesteia în funcția *show\_title*. În ambele cazuri, schimbările efectuate vor duce la erori de sintaxă. Variabilele externe amplifică problemele introduse de variabilele globale, deoarece pot fi schimbate în multe moduri, unele mai puțin evidente pentru programatorul care examinează codul. Poate chiar mai rău este faptul că, deoarece schimbările pot fi operate din diferite fișiere sursă, acestea pot fi detectate foarte greu.

Pentru a reduce vulnerabilitatea unei variabile globale, se poate reduce accesul la aceasta numai la fișierul sursă în care aceasta este definită. În acest caz, declarația variabilei este precedată de specificatorul *static*. De exemplu, următoarea instrucțiune restrânge domeniul variabilei *title* la fișierul SHOWTITL.CPP:

```
static char *title = "Succes cu C++";
```

Dacă adăugați specificatorul *static* declarației variabilei *title* din fișier și apoi recompilați cele două programe, editorul de legături va afișa un mesaj de eroare despre identificatorul *title* nerezolvat.

### INIȚIALIZAREA UNEI VARIABILE EXTERNE

Când o variabilă este precedată de cuvântul *extern*, avem o declarație, nu o definiție, deoarece compilatorul nu alocă memorie pentru variabilă. Aceasta se întâmplă numai în cazul când nu se inițializează variabila. De exemplu, următorul program, NOEXTERN.CPP, inițializează variabila externă *title* folosind șirul "Salvat de C++".

Datorită inițializării, compilatorul va alocă memorie pentru variabilă. La compilarea și legarea acestui program cu SHOWTITL.CPP, se va afișa:

```
C:\> NOEXTERN <ENTER>
The title is Rescued by C++
The title of this book is Success with C++
```

După cum se vede, programul folosește două variabile *title* diferite. În funcție de editorul de legături folosit, se pot detecta identificatori dublu definiți.

### SĂ ÎNȚELEM MEMBRII DE TIP STATIC AI UNEI CLASE

Ați învățat că, la declararea obiectelor unei clase, fiecare obiect își preia propriile sale variabile membru. De exemplu, dacă un program folosește două sau mai multe obiecte de tipul *date*, fiecare obiect va avea propriile sale variabile *month*, *day* și *year*. În funcție de scopul programului, pot apărea situații când două sau mai multe obiecte partajează aceeași variabilă membru. De exemplu, să presupunem că aveți obiecte de tipul *Nuke*, fiecare dintre ele destinat detecției unei anumite situații într-un reactor nuclear. Apoi, să presupunem că atunci când apare o condiție de eroare, obiectul setează variabila membru *melt\_down* la 1. Prin partajarea variabilei între obiectele *Nuke*, toate obiectele programului pot fi simultan la curent cu o situație periculoasă. Pentru a partaja o variabilă membru între mai multe obiecte, trebuie precedată declarația acelei variabile de cuvântul cheie *static*:

```
class Nuke {
public:
    Nuke(char *name, int limit);
    void show_status(void);
    void set_melt_down(void);
private:
    static int melt_down;
    char name[64];
    int limit;
};
```

Obiectele de tipul *Nuke* vor partaja variabilele membru *melt\_down*, dar vor avea fiecare copii ale variabilelor membru *name* și *limit*. În acest exemplu, variabila membru este declarată *private*, dar C++ suportă pe deplin variabile membru de tip *private* și *public* partajate. Apoi, după declarația clasei, variabila membru statică trebuie redeclarată și eventual inițializată astfel:

```
int Nuke::melt_down = 0;
```

Următorul program, MELTDOWN.CPP, ilustrează folosirea variabilei membru partajate:

```
#include <iostream.h>
#include <string.h>
```

```
class Nuke {
public:
    Nuke(char *name, int limit);
    void show_status(void);
    void set_melt_down(void);
private:
    static int melt_down;
    char name[64];
    int limit;
};

int Nuke::melt_down = 0;

Nuke::Nuke(char *name, int limit)
{
    strcpy(Nuke::name, name);

    Nuke::limit = limit;
}

void Nuke::show_status(void)
{
    cout << "Name: " << name;
    cout << " limit: " << limit;
    cout << " meltdown: " << melt_down << endl;
}

void Nuke::set_melt_down(void)
{
    melt_down = 1;
}

void main(void)
{
    Nuke rods("Check Rods", 1001);
    Nuke fission("Check Fission", 2002);

    rods.show_status();
    fission.show_status();

    rods.set_melt_down();

    rods.show_status();
    fission.show_status();
}
```

• La compilarea și execuția acestui program, pe ecran se va afișa:

```
C:\> MELTDOWN <ENTER>
Name: Check Rods limit: 1001 meltdown: 0
Name: Check Fission limit: 2002 meltdown: 0
```

```
Name: Check Rods limit: 1001 meltdown: 1
Name: Check Fission limit: 2002 meltdown: 1
```

După cum se observă, când obiectul *rods* determină valoarea variabilei partajate *melt\_down* la 1, obiectul *fission* observă imediat schimbarea.



### PARTAJAREA UNEI VARIABILE MEMBRU A UNUI OBIECT

Există situații când două sau mai multe obiecte trebuie să partajeze aceeași variabilă membru. O astfel de variabilă membru partajată se poate crea scriind în fața acesteia cuvântul cheie *static*. De exemplu, în clasa *Employee* prezentată mai jos, obiectele partajează variabilele *company\_name* și *health\_plan*:

```
class Employee {
public:
    Employee(char *name, long identifier, int job);
    void show_employee(void);
private:
    char name[64];
    long identifier;
    int job;
    static char company_name[64];
    static char health_plan[25];
};
```

Apoi, în afara declarației clasei, trebuie specificate declarațiile pentru variabilele partajate, ca mai jos:

```
char Employee::company_name[64];
char Employee::health_plan[25];
```

Fiecare obiect de tipul *Employee* creat de program va partaja aceste două variabile.

### FUNCȚII MEMBRU ÎN INTERIORUL ȘI ÎN EXTERIORUL DEFINIȚIEI CLASEI

La declararea unei clase, funcțiile membru pot fi declarate ca *funcții inline*, când instrucțiunile lor apar în interiorul declarației clasei, sau în afara acestei declarații. De exemplu, programul anterior MELTDOWN.CPP a folosit funcții membru în afara definiției clasei, cum este constructorul *Nuke* de mai jos:


```
Nuke::Nuke(char *name, int limit)
{
```

```
strcpy(Nuke::name, name);
Nuke::limit = limit;
}
```

Funcția constructor s-ar fi putut declara și *inline*, astfel:

```
class Nuke {
public:
    Nuke(char *name, int limit) {
        strcpy(Nuke::name, name);
        Nuke::limit = limit;
    }
    void show_status(void);
    void set_melt_down(void);
private:
    static int melt_down;
    char name[64];
    int limit;
};
```

Când se declară o funcție *inline*, fiecare obiect va primi un exemplar al codului funcției. La funcții declarate în afara clasei, pe de altă parte, fiecare obiect partajează aceeași copie a funcției. În majoritatea cazurilor, se recomandă a se folosi avantajul partajării codului pentru a reduce necesarul de memorie a programului.



**CHEIA SUCCESULUI**

**PARTAJAREA CODULUI UNEI FUNCȚII MEMBRU**

La declararea funcțiilor membru ale clasei, există două posibilități. Se pot declara funcții în cadrul definiției clasei (ca funcții *inline*) sau în afara acesteia. Avantajul declarării instrucțiunilor unei funcții membru în afara definiției clasei constă în faptul că fiecare obiect nou creat partajează același exemplar din codul programului. Prin urmare, dacă sunt create 100 de obiecte, se creează numai o singură funcție, pe care obiectele o vor partaja. În acest fel se reduce și necesarul de memorie a programului.

### SĂ REVEDEM OPERATORUL DE REZOLUȚIE GLOBALĂ

Este posibil ca într-un program numele variabilelor locale să coincidă cu cel al variabilelor globale. În astfel de situații, C++ asociază toate referințele la numele variabilei locale. De exemplu, următorul program, `USELOCAL.CPP`, definește variabila globală *number*, atribuind variabilei valoarea 1001. În funcția *main*, programul folosește variabila locală *number* în cadrul unui ciclu *for* producând un conflict de nume. La afișarea valorii variabilei *number*, programul va afișa valoarea variabilei locale, ascunzând astfel variabila globală.

```
#include <iostream.h>

int number = 1001; // Declare global variable

void main(void)
{
    for (int number = 1; number < 5; number++)
        cout << number << endl;

    cout << "Value of number is " << number << endl;
}
```

La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> USELOCAL <ENTER>
1
2
3
4
Value of number is 5
```

După cum se observă, programul a folosit variabila locală *number*, ignorând variabila globală. Dacă doriți totuși folosirea variabilei globale, puteți scrie înaintea numelui variabilei operatorul de rezoluție globală (*::*). Următorul program, `USEGLOB.CPP`, folosește operatorul de rezoluție globală pentru a afișa valoarea variabilei globale *number* după terminarea ciclului *for*:

```
#include <iostream.h>

int number = 1001; // Declare global variable

void main(void)
{
    for (int number = 1; number < 5; number++)
        cout << number << endl;

    cout << "Value of number is " << ::number << endl;
}
```

La compilarea și execuția programului, pe ecran vor apărea rezultatele:

```
C:\> USEGLOB <ENTER>
1
2
3
4
Value of number is 1001
```

Așa cum se observă, programul afișează valoarea variabilei globale *number* la sfârșitul ciclului. Când se folosește operatorul de rezoluție globală în acest mod, C++ va referi întotdeauna variabila globală.

Să considerăm, de exemplu, următorul program, WHICHNUM.CPP, care folosește 3 variabile diferite denumite *number*:

```
#include <iostream.h>

int number = 1001; // Global variable

void main(void)
{
    for (int number = 1; number < 5; number++)
    {
        int number = 1;

        cout << "local number " << number << " :: number is " <<
            ::number << endl;
    }
}
```

Se observă că în blocul de adâncime maximă programul folosește operatorul de rezoluție globală. Variabila referită cu acest operator este variabila globală, și nu variabila *number* definită în următorul ciclu exterior blocului. La compilarea și execuția acestui program, ecranul va afișa următoarele:

```
C:\> WHICHNUM <ENTER>
local number 1 :: number is 1001
local number 1 :: number is 1001
local number 1 :: number is 1001
local number 1 :: number is 1001
```

Așa cum probabil ați observat, folosirea variabilelor cu același nume în scopuri diferite poate duce la programe greu de înțeles. Pentru a evita astfel de confuzii, asigurați-vă că folosiți nume sugestive pentru fiecare variabilă în parte. În majoritatea cazurilor (cu excepția variabilelor contor sau index), folosirea unor nume sugestive va elimina multe conflicte de nume. În al doilea rând, evitați pe cât posibil folosirea variabilelor globale. În sfârșit, dacă asemenea conflicte apar totuși, asigurați-vă că ați înțeles bine regulile de domeniu din C++, ca și durata de viață a variabilelor, ce va fi discutată în continuare.

### SĂ ÎNȚELEM DURATA DE VIAȚĂ A UNUI IDENTIFICATOR

*Durata de viață* a unei variabile definește perioada de timp în care variabila rămâne în domeniu. În mod normal, viața unei variabile începe când aceasta este definită și se termină când iese din domeniu. De exemplu, când într-o funcție se declară variabile locale, acestea intră în acțiune la apelarea funcției și

își încheie existența la terminarea funcției. Analog, dacă un program definește o variabilă globală, durata de viață a acesteia începe la definiție și continuă până la terminarea programului.

După discuția prezentată în acest capitol, se poate crede că durata de viață a unei variabile este o chesiuie simplă. Din păcate, folosirea calificatoarelor, cum este *static*, poate complica conceptul duratei de viață. De exemplu, următoarea funcție, *use\_count*, folosește variabila statică *counter* pentru a calcula numărul de apeluri ale funcției:

```
long use_count(void)
{
    static long counter = 0;

    // Statements

    return(counter);
}
```

În acest exemplu, durata de viață a variabilei *counter* începe la apelarea funcției, care la rândul său va inițializa și variabila. Spre deosebire de alte variabile locale declarate în interiorul funcției, a căror durată de viață se încheie o dată cu terminarea funcției, durata de viață a variabilei *counter* va continua până la terminarea programului.

Așa cum ați învățat, C++ permite declararea variabilelor în orice parte a programului. Următoarea instrucțiune *if*, de exemplu, declară o variabilă contor statică.

```
if (some_condition)
{
    static int if_counter = 0;

    // Statements

    if_counter++;
}
```

Pentru a îmbunătăți performanța programului, se poate folosi o astfel de variabilă pentru a determina de câte ori se execută o anumită parte a codului. În acest caz, durata de viață a variabilei începe o dată cu prima execuție a instrucțiunii *if* și se încheie la terminarea programului.

Un obicei frecvent folosit în programele C++ este de a declara o variabilă în interiorul unui ciclu *for*, ca mai jos:

```
for (int i = 0; i < 10; i++)
    some_operation();
```

Când se declară o variabilă în acest mod, durata de viață a variabilei începe când ciclul este executat pentru prima dată și se încheie când blocul care conține ciclul este încheiat. Următoarea secvență definește variabilele *i* și *j* în cadrul a două cicluri *for* incluse:

```
for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
        cout << i << " " << j << endl;
```

La execuția acestui fragment de program, ecranul va afișa:

```
01
02
03
10
11
12
20
21
22
```

După cum se observă, programul tratează corect variabilele *i* și *j*. La prima vedere, se poate crede că secvența de cod creează variabilele *i* și *j*, care vor continua să existe până la încheierea blocului de program în care sunt scrise cele două cicluri. Dar, așa cum se va vedea, variabila *i* există până la sfârșitul blocului curent, în timp ce variabila *j* își încetează existența la terminarea ciclului *for* exterior. Următorul program, SYNTAX.CPP, încearcă să afișeze valorile ambelor variabile după încheierea ciclurilor. Dacă încercați să compilați acest program, compilatorul va genera o eroare de sintaxă, semnalând faptul că variabila *j* este necunoscută:

```
#include <iostream.h>

void main(void)
{
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            cout << i << " " << j << endl;

    cout << "Ending values i: " << i << " j: " << j << endl;
}
```

Identificator j necunoscut

Când lucrați cu mai multe variabile, încercați să determinați durata de viață a fiecăreia. Înțelegând mai bine domeniul și durata de viață a unei variabile, veți înțelege mai bine operațiile de finețe pe care le realizează compilatorul, având astfel posibilitatea de a scrie un cod mai eficient.

## SĂ ÎNȚELEGEM EDITAREA LEGĂTURILOR

La programele mici, cum sunt cele prezentate în această carte, de obicei tot codul este inclus într-un singur fișier sursă. Pe măsură ce programele devin mai mari, veți înțelege mai ușor codul programului dacă acesta este împărțit logic în mai multe fișiere. În plus, se poate reduce timpul de dezvoltare a programului prin compilarea separată a componentelor acestuia. În acest mod, dacă se

efectuează schimbări asupra unei anumite părți a programului, va trebui recompilat un singur fișier sursă.

În continuare, se poate folosi editorul de legături pentru a lega fișierele obiect compilate anterior. În Capitolul 17 veți învăța cum să construiți o bibliotecă de clase. Deocamdată, însă, veți compila și apoi lega două fișiere sursă înrudite. Să creăm mai întâi fișierul STR\_LEN.CPP ce conține următoarea funcție:

```
int str_len(char *string)
{
    int length = 0;

    while (*string++)
        length++;

    return(length);
}
```

Prin compilarea acestui fișier se creează fișierul cu cod obiect STR\_LEN.OBJ. Dacă se folosește compilatorul Borland C++, este necesară următoarea comandă:

```
C:\> BCC -c STR_LEN.CPP <ENTER>
```

**Observație:** Opțiunea "-c" din linia de comandă indică compilatorului să efectueze o compilare, fără a executa și legarea. Fără această opțiune pe linia de comandă compilatorul va încerca să creeze fișierul STR\_LEN.EXE (litera din opțiunea "-c" trebuie să fie literă mică).

Apoi se creează fișierul HOWLONG.CPP care folosește funcția anterioară:

```
#include <iostream.h>

int str_len(char *);

void main(void)
{
    char *string = "Success with C++";

    cout << "The string " << string << " contains " <<
        str_len(string) << " characters" << endl;
}
```

Prin compilarea acestui program se obține fișierul obiect HOWLONG.OBJ:

```
C:\> BCC -c HOWLONG.CPP <ENTER>
```

Cele două fișiere obiect se pot combina cu ajutorul editorului de legături, formându-se programul executabil HOWLONG.EXE. Dacă folosiți Borland C++, se poate realiza legarea celor două fișiere cu ajutorul comenzii:

```
C:\> BCC HOWLONG.OBJ STR_LEN.OBJ <ENTER>
```

**Observație:** Multe compilatoare, printre care și Borland C++, permit specificarea în linia de comandă a fișierelor obiect ce urmează a fi compilate și apoi legate. Astfel, dacă folosiți Borland C++ se poate compila programul HOWLONG.CPP și apoi lega fișierul STR\_LEN.OBJ folosind următoarea comandă:

```
C:\> BCC HOWLONG.CPP STR_LEN.OBJ <ENTER>
```

Dacă examinați programul HOWLONG.CPP, veți vedea că el include prototipul funcției `str_len`. Chiar dacă funcția `str_len` se află în alt fișier, ea trebuie totuși declarată înainte de a fi folosită.

Când programul folosește mai multe fișiere sursă sau obiect, trebuie înțelese legăturile (referințele între entitățile programului) care sunt folosite de compilator și de editorul de legături pentru a rezolva referințele între mărimile programului. C++ definește două tipuri de legături, interne și externe. Când un nume de identificator este local unui fișier sursă, identificatorul are *legătură internă*. Dacă domeniul unui identificator depășește fișierul în care este declarat, identificatorul are *legătură externă*. Să presupunem, de exemplu, că un fișier sursă definește variabila globală `error_number` ca mai jos:

```
hms drnr_mtladr: .. Glnaa1
unhc lahm(unhc(
  z
  .. Ssasdldmsr
})
```

În acest caz, domeniul variabilei globale depășește fișierul sursă, prin urmare variabila are legătură externă. De exemplu, să creăm următorul program ESSOSNUM.CPP:

```
unhc rgnv_drrnr_mtladr(unhc(
hms drnr_mtladr = 1001:
unhc lahm(unhc(
  z
  rgnv_drrnr_mtladr((:
})
```

După cum se poate vedea, programul atribuie o valoare variabilei globale și apoi apelează funcția `show_error_number` pentru a afișa valoarea variabilei. Ca și înainte, programul include un prototip pentru funcție, chiar dacă funcția este declarată în alt fișier.

Să creăm acum fișierul SHOWESS.CPP, care definește funcția `show_error_number`:

```
#hmb1tcd <hnrscdal-g>
dxsdrm hms drnr_mtladr:
unhc rgnv_drrnr_mtladr(unhc(
  z
```

```
cout << "Error number: " << error_number << endl;
}
```

În acest caz, fișierul folosește declarația *extern* pentru a informa compilatorul că variabila `error_number` este definită în alt fișier. După compilare, editorul de legături trebuie să rezolve referința la variabilă.

Când se scriu programe multifîșier ce se bazează pe astfel de legături externe, trebuie specificat corect tipul variabilei externe. În majoritatea cazurilor, compilatorul nu poate detecta o variabilă externă de tip greșit declarat, ceea ce duce la o eroare foarte dificil de detectat. De exemplu, următorul fișier, WRONGERR.CPP declară în mod incorect tipul variabilei externe `error_number` ca fiind `float`:

```
#include <iostream.h>
extern float error_number; _____ Specificare de tip incorectă
void show_error_number(void)
{
  cout << "Error number: " << error_number << endl;
}
```

Următorul fișier, BAD\_INIT.CPP, atribuie o valoare inițială variabilei externe `error_number`. Din păcate, atunci când compilatorul realizează inițializarea, declarația devine o definiție, ascunzând variabila globală externă anterioară:

```
#include <iostream.h>
extern int error_number = 2002;
void show_error_number(void)
{
  cout << "Error number: " << error_number << endl;
}
```

Dacă se compilează și se leagă programele BAD\_INIT.CPP și ERRORNUM.CPP, se va afișa pe ecran următorul mesaj:

```
C:\> ERRORNUM <ENTER>
Error number: 2002
```

În funcție de editorul de legături folosit, identificatorul dublu definit poate fi detectat. În caz contrar, depanarea erorii este o sarcină foarte dificilă. Pentru a înțelege mai bine prelucrarea realizată, s-ar putea modifica programul ERRORNUM.CPP pentru a afișa valoarea a lui `error_number` astfel:

```
#include <iostream.h>
void show_error_number(void):
```



```
int error_number = 1001;

void main(void)
{
    show_error_number();
    cout << "In main: " << error_number << endl;
}
```

În acest caz, ieșirea programului devine:

```
C:\> ERRORNUM <ENTER>
Error number: 2002
In main 1001
```

Așa cum se folosesc membrii de tip *private* ai unei clase pentru a nu putea fi apelați din afara obiectului, se poate limita domeniul unei variabile sau funcții la fișierul sursă curent. Dacă în fața numelui unei variabile globale se scrie cuvântul cheie *static*, atunci legarea acelei variabile devine *internă* fișierului sursă. Cu alte cuvinte, variabila nu mai poate fi utilizată în afara fișierului sursă curent. Următorul program, INTERNAL.CPP, definește două funcții denumite *internal* și *external*. Deoarece funcția *internal* este precedată de cuvântul *static*, domeniul funcției este local fișierului sursă, pe când funcția *external* are legătură externă:

```
#include <iostream.h>

static void internal(char *message)
{
    cout << message << endl;
}

void external(char *message)
{
    cout << message << endl;
}

void some_function(void);

void main(void)
{
    internal("Using the internal function");
    external("Using the external function");

    some_function();
}
```

Să observăm că în program se folosește și funcția *some\_function*, care este declarată în afara fișierului sursă. Următorul program, SOMEFUNC.CPP definește *some\_function*:

```
void external(char *):
void some_function(void)
{
```

```
external("Using external from outside the source file");
}
```

După cum se vede, funcția *some\_function* apelează funcția *external* pentru a afișa un mesaj. După compilarea și legarea celor două fișiere, se lansează în execuție programul INTERNAL care va tipări:

```
c:\> INTERNAL <Enter>
Using the internal function
Using the external function
Using external from outside the source file
```

Dacă se modifică fișierul SOMEFUNC.CPP pentru a folosi funcția *internal*, editorul de legături va afișa un mesaj de eroare referitor la nerezolvarea referinței la funcția *internal*. Deoarece funcția *internal* are o legătură internă, ea nu poate fi folosită în afara fișierului sursă.

La proiectarea programelor multifîșier și a bibliotecilor de programe, trebuie avută în vedere legarea referințelor variabilelor și funcțiilor. Așa cum ați văzut, erorile din timpul legării referințelor pot fi la fel de greu de detectat, ca și erorile de domeniu ale entităților programului.

## REZUMAT

Acest capitol a analizat regulile domeniului de vizibilitate a mărimilor programului. Dacă încă nu vă este totul clar, aveți răbdare și exersați cu programele prezentate, observând influența modificărilor efectuate. În Capitolul 10 veți examina referințele C++, care oferă un nume alternativ pentru o variabilă. Înainte de a continua cu Capitolul 10, asigurați-vă că ați învățat următoarele:

- ✓ Domeniul unui identificator definește partea din program în care identificatorul poate fi folosit.
- ✓ Înainte de folosire, un identificator trebuie declarat. În general, o declarație specifică un tip și un nume. Dacă la declarații se alocă și memorie, atunci declarația se consideră o definiție.
- ✓ C++ acceptă 4 tipuri de domenii: local, de funcție, de fișier, de clasă.
- ✓ Domeniul local corespunde mărimilor definite într-un bloc (între acolade).
- ✓ Domeniul de funcții corespunde numai etichetelor de salt. O instrucțiune *goto* nu poate efectua saltul la o etichetă în afara funcției curente (*goto* este permis numai local).
- ✓ Domeniul de fișier corespunde variabilelor globale și identificărilor de funcții care sunt cunoscuți în tot fișierul sursă (sau în mai multe fișiere).
- ✓ Domeniul de clasă corespunde variabilelor membru și funcțiilor membru care sunt accesibile numai obiectelor clasei.

- ✓ C++ permite declararea variabilelor intercalată cu instrucțiunile programului, nu numai la începutul unui bloc. În multe cazuri, declararea unei variabile aproape de locul de folosire îmbunătățește claritatea programului.
- ✓ Dacă numele unei variabile locale intră în conflict cu cel al unei variabile globale sau al unei variabile membru a unei clase, compilatorul va asocia referințele la acel nume cu variabila locală. Pentru a accesa variabila globală sau variabila clasei, trebuie folosit operatorul de rezoluție globală (::).
- ✓ Durata de viață a unei variabile definește perioada în timpul căreia variabila rămâne în domeniu. În mod normal, existența unei variabile începe în momentul definirii ei.
- ✓ C++ acceptă legături interne și externe ale entităților unui program. Identificatorii cu legătură internă sunt cunoscuți numai în fișierul sursă în care sunt definiți. Identificatorii cu legătură externă pot fi apelați din diferite fișiere sursă.

## CAPITOLUL 10

### ACOMODAREA CU REFERINȚELE C++

Dacă sunteți un programator C experimentat, probabil că vă descurcați fără probleme cu operațiile cu pointeri. De aceea ar trebui să fiți pregătiți pentru a înțelege *referințele C++*, care sunt un fel de adrese, dar și un fel de valori. Acest capitol examinează referințele C++ în detaliu. Totuși, în majoritatea cazurilor, veți continua să folosiți operațiile standard cu pointeri, așa cum ați făcut și până acum.

Programatorii sunt familiari cu majoritatea operațiilor cu pointeri. Însă apar situații când lucrați cu variabile complexe, cum sunt structurile și obiectele, când folosirea referințelor este foarte eficientă. În plus, în Capitolul 11 veți învăța să creați proprii manipulatori I/O, cum sunt *hex* sau *endl*. Așa cum veți vedea, manipulatorii lucrează cu referințe la anumite stream-uri I/O.

La terminarea acestui capitol, veți cunoaște:

- ♦ Ce este o referință
- ♦ Capcanele numelor asociate
- ♦ Ce trebuie făcut pentru a crea o referință
- ♦ Ce operații se pot efectua cu referințe
- ♦ Cum vede compilatorul C++ o referință
- ♦ Folosirea referințelor
- ♦ Ce sunt obiectele ascunse

Dacă se cunosc funcțiile care primesc și returnează referințe, manipulatorii de stream I/O vor fi ușor de creat.

#### O REFERINȚĂ ESTE UN NUME ASOCIAT

Așa cum cunoașteți, un nume de variabilă corespunde unei locații de memorie ce este accesibilă folosind o anumită adresă. O *referință C++* este un al doilea nume (sau *alias*) pentru o locație de memorie. Înainte de a folosi o referință, aceasta trebuie declarată în program. Ca și alte declarații, o referință are un

nume și un tip. Când declarați o referință, trebuie să atribuiți referinței variabila pentru care referința servește drept nume asociat (*alias*). Declarațiile de referințe plasează caracterul ampersand (&) imediat după tipul variabilei, ca de exemplu *int&*. Următoarele instrucțiuni, de exemplu, declară variabila *alias* ca o referință la variabila *count*:

```
int count;
```

```
int& alias = count; // Create the reference
```

Diagram illustrating the reference declaration:

- int&*: Referință de tip int
- alias*: Nume referință (alias)
- = count*: Variabilă la care referința este alias

Următorul program, FIRSTREF.CPP, creează o referință denumită *alias* pentru variabila *count*. Programul afișează valoarea lui *count* folosind atât variabila, cât și referința. Programul incrementează apoi valoarea lui *count* incrementând variabila *alias*:

```
#include <iostream.h>

void main(void)
{
    int count = 1000;

    int& alias = count; // Create the reference

    cout << "count's value is " << count << " and " << alias << endl;

    alias++; // Increment count's value

    cout << "count's value is " << count << " and " << alias << endl;
}
```

O referință nu este altceva decât un al doilea nume (*alias*) pentru o variabilă. După crearea unei referințe, programul poate folosi fie referința, fie variabila, pentru a avea acces la aceeași locație de memorie. În exemplul precedent, programul nu numai a afișat valoarea lui *count* folosind referința, dar a folosit referința și pentru incrementarea valorii acesteia.

La compilarea și execuția acestui program pe ecran va apărea:

```
C:\> FIRSTREF <ENTER>
count's value is 1000 and 1000
count's value is 1001 and 1001
```

Se pot crea referințe pentru variabile de orice tip (*float*, *long* sau chiar *struct*). Următorul program, REFTYPES.CPP, de exemplu, creează referințe pentru diferite tipuri de date și le folosește pentru manevrarea valorilor variabilelor asociate:

```
#include <iostream.h>

void main(void)
{
    float pi = 22.0 / 7.0;

    float& pi_alias = pi;

    struct Date { int month; int day; int year; }
        today = { 9, 30, 94 };

    Date& birthday = today;

    cout << "Pi is " << pi << " or " << pi_alias << endl;

    cout << "My birthday is " << birthday.month << '/' <<
        birthday.day << '/' << birthday.year << endl;
}
```

După cum se observă, programul creează două referințe, una de tipul *float* și alta de tipul *Date*. La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> REFTYPES <ENTER>
Pi is 3.142857 or 3.142857
My birthday is 9/30/94
```

Scopul acestor două prime programe este acela de a arăta că o referință este pur și simplu un alias sau al doilea nume al unei variabile. În mod normal, referințele nu se folosesc în maniera indicată aici. Folosind două sau mai multe nume pentru aceeași locație de memorie, programele devin mai greu de înțeles. Cel mai frecvent, referințele se folosesc pentru transferul de valori către funcții și dinspre funcții, în special pentru valorile complexe de tip structuri sau obiecte.

#### CHEIA SUCCESULUI



#### CREAREA UNEI REFERINȚE

O referință este un nume asociat (sau alias) pentru o variabilă. Referințele au nume și tip (ca *int*, *float*, *class* sau *struct*). Pentru a crea o referință, ea trebuie declarată, similar cu modul în care declarăm variabila. De exemplu, următoarele instrucțiuni creează o referință denumită *hillary* pentru o variabilă de tipul *president* denumită *bill*:

```
struct president { char name[64]; int age; short qualifications;
} bill;
```

```
president& hillary = bill;
```

Spre deosebire de o variabilă, totuși, valoarea atribuită unei referințe nu poate fi schimbată. O dată ce ați atribuit o valoare unei referințe, această valoare rămâne asociată pentru toată durata execuției blocului de program.

## O REFERINȚĂ NU ESTE O VARIABILĂ

Deși o referință seamănă cu o variabilă, în sensul că trebuie specificate un nume și un tip și trebuie atribuită o valoare, o referință nu este o variabilă. O dată ce ai atribuit o valoare unei referințe, referința nu se poate schimba. Cu alte cuvinte, o dată ce ai asociat un nume unei variabile, nu poți folosi același nume și pentru o altă variabilă.

Următorul program REF\_FUNC.CPP, folosește o variabilă referință la un parametru al funcției. Când se folosește o referință locală în acest mod, ea rămâne validă pe durata funcției. De fiecare dată când se apelează funcția, referința va corespunde noului parametru transmis. Funcția nu poate modifica valoarea referinței:

```
#include <iostream,h>

void use_reference(int value)
{
    int& alias = value;

    cout << "The parameter value using alias is " << alias << endl;
}

void main(void)
{
    for (int i = 0; i < 5; i++)
        use_reference(i);
}
```

La compilarea și execuția programului, pe ecran va apărea:

```
C:\> REF_FUNC <ENTER>
The parameter value using alias is 0
The parameter value using alias is 1
The parameter value using alias is 2
The parameter value using alias is 3
The parameter value using alias is 4
```

Așa cum se observă, referința este asignată noului parametru de fiecare dată când funcția este invocată.

## POSIBILE SINTAXE PENTRU REFERINȚE



Toate referințele din programele prezentate în această carte plasează ampersand-ul (&) imediat după numele tipului, ca mai jos:

int value;

int& alias = value;

C++ vă permite, totuși, să plasați ampersand-ul între numele tipului și numele referinței, sau chiar imediat după numele referinței. Următoarele declarații, prin urmare, sunt identice:

int& alias = value;      int & alias = value;      int &alias = value;

Cu toate că puteți întâlni toate cele 3 formate anterioare, este bine să alegeți unul din ele și să îl folosiți constant. În acest mod programele vor deveni mai consecvente (uniforme) în notații, și deci mai ușor de înțeles.

## REGULI PENTRU LUCRUL CU REFERINȚE

Așa cum ai învățat, o referință creează un al doilea nume pentru o variabilă. Multe din operațiile care se pot efectua cu variabila, se pot efectua și cu referința ei. De exemplu, se pot efectua operații aritmetice, de atribuire, se poate afișa valoarea variabilei, se poate chiar testa valoarea în operatori condiționali. Rămân, însă, o serie de operații care nu pot fi efectuate pe referința unei variabile.

Programatorul nu are capacitatea de a:

- obține adresa unei referințe folosind operatorul de adresă (&)
- atribui un pointer unei referințe
- compara două valori referință
- crea deplasamentul (offset) unei referințe
- modifica valoarea unei referințe

Să nu înțelegeți greșit regula despre folosirea unei referințe într-o operație condițională. De exemplu, următorul program, TESTREF.CPP, folosește referința *alias* într-o instrucțiune *if* pentru a testa dacă valoarea din locația de memorie referită de *alias* este 1001:

```
#include <iostream.h>

void main(void)
{
    int value = 1001;

    int& alias = value;

    if (alias == 1001)
        cout << "alias contains the value 1001" << endl;
    else
        cout << "alias does not the value 1001" << endl;
}
```

După cum se observă, programul folosește referința într-un test condițional. Dar un asemenea test nu se poate folosi pentru a determina dacă două referințe adresează aceeași locație de memorie. În schimb, se pot compara valorile din fiecare adresă de memorie corespunzătoare referințelor. De exemplu, următorul program, TEST2REF.CPP, folosește o instrucțiune *if* pentru a compara valorile referite de *alias\_one* și *alias\_two*:

```
#include <iostream.h>

void main(void)
{
    int one = 1;
    int& alias_one = one;

    int uno = 1;
    int& alias_two = uno;

    if (alias_one == alias_one)
        cout << "The references contain the same value " <<
            alias_one << endl;
    else
        cout << "The alias values differ " << alias_one << " and " <<
            alias_two << endl;
}
```

Așa cum se observă, instrucțiunea *if* compară valorile celor două referințe. Dacă compilați și executați acest program, pe ecran va apărea:

```
C:\> TEST2REF <ENTER>
The references contain the same value 1
```

Se observă că instrucțiunea *if* testează valorile conținute în locațiile de memorie asociate referințelor, și nu adresele de memorie corespunzătoare fiecărei referințe. În acest exemplu, cele două referințe nu sunt dependente. Întâmplător, locațiile de memorie corespunzătoare referințelor conțin valoarea 1.

## REFERINȚA ESTE O VALOARE SAU O ADRESĂ?

Mulți programatori începători în lucrul cu referințele încearcă să le identifice cu adresele. Cea mai bună imagine a unei referințe este o *adresă rezolvată*. Pentru compilatorul C++, o referință este o adresă. Când compilatorul generează codul obiect, referințele sunt tratate drept adrese. În cadrul programului, însă, referința corespunde unei valori ce se află la o anumită adresă. Cu alte cuvinte, o referință permite programului să manipuleze o valoare dintr-o locație de memorie, fără a fi nevoie de a lucra cu pointeri. Orice operație care poate fi efectuată cu referințe poate fi efectuată și cu pointeri.

Referințele sunt ușor de folosit în cadrul programului, deoarece compilatorul este cel care rezolvă, în fundal, problemele legate de adrese și valori. Întrucât referințele nu sunt nici variabile, nici pointeri, există anumite operații pe care acestea nu le acceptă.

O modalitate de a obține o imagine reală asupra modului de manevrare a referințelor de către compilator este de a genera listingul în limbaj de asamblare al programului. Dacă folosiți compilatorul Borland C++, un astfel de listing se obține folosind comutatorul *-S*, ca mai jos:

```
C:\> BCC -S FILENAME.CPP <ENTER>
```

Sintaxa pentru compilatorul Microsoft Visual C++ este următoarea:

```
C:\> CL /Fa /Fs FILENAME.CPP <ENTER>
```

**Observație:** La ambele compilatoare, comutatorii din linia de comandă sunt sensibili la diferența dintre majuscule și literele mici.

După ce obțineți listingul în limbaj de asamblare, modificați conținutul fișierului și observați cu atenție instrucțiunile folosite de compilator pentru rezolvarea adreselor.

## FOLOSIREA REFERINȚELOR DREPT PARAMETRI

Referințele dau posibilitate programelor să beneficieze de avantajele pointerilor, fără o sintaxă complicată (cu alte cuvinte, nu trebuie transferate adresele parametrilor și apoi folosiți pointerii pentru a obține valorile acestora). Următorul program, REFPARAM.CPP, de exemplu, transferă o referință funcției *change\_value*. În cadrul funcției, se folosește un parametru de tip referință pentru a se putea schimba valoarea variabilei asociată acestui parametru:

```
#include <iostream.h>

void change_value(int& value)      Parametru referință
{
    ...
}
```

```

    value = 1001;
}

void main(void)
{
    int count = 1;

    int& alias = count;

    cout << "count's value is " << count << " or " << alias << endl;

    change_value(alias);      Transferul referinței ca parametru

    cout << "count's value is " << count << " or " << alias << endl;
}

```

Se observă că programul transferă referința *alias* funcției *change\_value*. Funcția folosește o altă referință, denumită *value*. La compilarea și execuția acestui program, pe ecran va apărea:

```

C:\> REFPARAM <ENTER>
count's value is 1 or 1
count's value is 1001 or 1001

```

Este evident că programul elimină nevoia de a lucra cu pointeri și adrese. Totuși, programul introduce un nume suplimentar (în acest caz *alias*) pe care programatorul trebuie să-l rețină. De regulă, cu cât sunt mai puține nume de variabile și referințe de memorat de către utilizator, cu atât programul va fi mai ușor de citit și modificat.

Folosind pointeri, se poate obține același rezultat, cum se arată în programul următor, PTRPARAM.CPP:

```

#include <iostream.h>

void change_value(int *value)
{
    *value = 1001;      Argument pointer
}

void main(void)
{
    int count = 1;

    cout << "count's value is " << count << endl;

    change_value(&count);      Transferul unui argument prin adresă

    cout << "count's value is " << count << endl;
}

```

În mod similar, următorul program, REF\_EXCH.CPP, transferă două referințe funcției *exchange\_values*, care va schimba între ele valorile acestora:

```

#include <iostream.h>

void exchange_values(int &a, int& b)
{
    int temp;      // Not a reference      Doi parametri referință

    temp = a;
    a = b;
    b = temp;
}

void main(void)
{
    int one = 1;
    int two = 2;

    int& one_alias = one;
    int& two_alias = two;

    cout << "Before exchange: one is " << one << " and two is " <<
        two << endl;

    exchange_values(one_alias, two_alias);      Transferul referințelor ca parametri

    cout << "After exchange: one is " << one << " and two is " <<
        two << endl;
}

```

Ca și mai înainte, programul elimină nevoia de a lucra cu pointeri și adrese, dar adaugă două referințe (*one\_alias* și *two\_alias*). Când numărul variabilelor și referințelor folosite într-un program crește, va crește și gradul de dificultate al acestuia. Să observăm că, în cadrul funcției *exchange\_values*, variabila *temp* este declarată de tip *int*, și nu referință la tipul *int*. Rețineți că instrucțiunea ce urmează declarației variabilei *temp* atribuie acesteia valoarea conținută în locația de memorie asociată variabilei *a* (care este 1). Dacă funcția ar fi declarat ca o referință, atunci *temp* ar fi devenit un al doilea nume pentru variabila *a*:

```

int &temp = a;      // Create an alias

```

Exersați acest program pentru a vă asigura că ați înțeles de ce variabila *temp* nu trebuie să fie referință. Următorul program, PTR\_EXCH.CPP, realizează același schimb de valori, dar folosind pointeri. Din același motiv, variabila *temp* este declarată de tip *int*, și nu pointer la *int*:

```
#include <iostream.h>

void exchange_values(int *a, int *b)
{
    int temp;    // Not a pointer

    temp = *a;
    *a = *b;
    *b = temp;
}

void main(void)
{
    int one = 1;
    int two = 2;

    cout << "Before exchange: one is " << one << " and two is " <<
        two << endl;

    exchange_values(&one, &two);

    cout << "After exchange: one is " << one << " and two is " <<
        two << endl;
}
```

## FOLOSIREA REFERINTELOR CU PARAMETRII ȘI OBIECTELE UNEI STRUCTURI

O utilizare frecventă a referințelor este de a transfera parametri de structuri către funcții sau de a returna variabile complexe. De exemplu, următorul program, SHOWDATE.CPP, folosește funcțiile *get\_date* și *show\_date* pentru a cere utilizatorului data curentă și apoi pentru a o afișa. Programul memorează data într-o structură de tip *Date*, ca mai jos:

```
struct Date {
    int month;
    int day;
    int year;
};
```

Veți vedea că programul transferă o referință la o variabilă de tip *Date* funcției *get\_date* și chiar variabila de tipul *Date* funcției *show\_date*:

```
#include <iostream.h>

struct Date {
    int month;
    int day;
```

```
int year;
};

void get_date(Date& date)
{
    cout << "Type in today's month: ";
    cin >> date.month;

    cout << "Type in today's day: ";
    cin >> date.day;

    cout << "Type in today's year: ";
    cin >> date.year;
}

void show_date(Date date)
{
    cout << "Today's date is: " << date.month << '/' << date.day <<
        '/' << date.year << endl;
}

void main(void)
{
    Date today;

    Date& todays_date = today;

    get_date(todays_date);

    show_date(today);
}
```

Așa cum se observă, programul transferă o referință funcției *get\_date*, dar variabila de tip structură este transferată funcției *show\_date*. Deoarece funcția *get\_date* modifică membrii structurii, ea trebuie să lucreze fie cu pointeri, fie cu referințe la structură.

Folosind referințe, funcția elimină nevoia de a folosi operații cu pointeri, de felul următor:

```
void get_date(Date *date)
{
    cout << "Type in today's month: ";
    cin >> date->month;

    cout << "Type in today's day: ";
    cin >> date->day;
```

```
cout << "Type in today's year: ";
cin >> date->year;
}
```



### EXPLICAȚII OPERAȚIILE CU REFERINȚE

Majoritatea programatorilor C și C++ sunt familiarizați cu operațiile pe pointeri. Dacă alegeți referințele, explicați în detaliu aceste operații. Altfel, cei care citesc programul vor sesiza cu dificultate diferența dintre numele de variabile similare (variabila și referința ei). În cazul programului SHOWDATE.CPP anterior, comentariile la funcția *main* s-ar putea face astfel:

```
void main(void)
{
    Date today;

    Date& todays_date = today; // Create an alias for today that
                                // eliminates pointer operations
                                // Within functions that change
                                // structure members

    get_date(todays_date);      // Todays_date is a reference that
                                // aliases the today structure

    show_date(today);           // Pass the actual date structure--
                                // not the reference
}
```

Când o funcție returnează o dată de tip structură, aceasta se face cu un aport considerabil de memorie, deoarece membrii structurii trebuie copiați din stivă în noua structură, căreia i-au fost atribuiți. Să presupunem, de exemplu, că funcția *get\_date* returnează o dată de tipul *Date*, cum se arată în programul RET\_STRU.CPP:

```
#include <iostream.h>

struct Date {
    int month;
    int day;
    int year;
};

// Funcția returnează o structură de tip Date
Date get_date(void)
{
    Date date;
```

```
cout << "Type in today's month: ";
cin >> date.month;
```

```
cout << "Type in today's day: ";
cin >> date.day;
```

```
cout << "Type in today's year: ";
cin >> date.year;
```

```
return(date);
}

void show_date(Date date)
{
    cout << "Today's date is: " << date.month << '/' << date.day <<
        '/' << date.year << endl;
}
```

```
void main(void)
{
    Date today;

    today = get_date(); // Atribuirea valorii returnate

    show_date(today);
}
```

Dacă generați un listing în limbajul de asamblare pentru programul precedent, ați putea determina modul în care compilatorul manipulează stiva pentru a transfera și returna structurile.

Pentru a elimina surplusul de memorie impus de manipularea stivei, se poate folosi o referință la structură, cum se arată în următorul program, RET\_REF.CPP:

```
#include <iostream.h>

struct Date {
    int month;
    int day;
    int year;
};

Date& get_date(Date &date)
{
    cout << "Type in today's month: ";
    cin >> date.month;

    cout << "Type in today's day: ";
    cin >> date.day;
```



```

    cout << "Type in today's year: ";
    cin >> date.year;

    return(date);
}

void show_date(Date& date)
{
    cout << "Today's date is: " << date.month << '/' << date.day <<
        '/' << date.year << endl;
}

void main(void)
{
    Date today;

    Date& todays_date = today; // Alias for today

    todays_date = get_date(todays_date);

    show_date(todays_date);
}

```

Așa cum se observă, ambele funcții, *get\_date* și *show\_date*, lucrează cu referințe la structuri. În consecință, programul capătă un plus de consecvență. În plus, se poate modifica funcția *main* prin combinarea a două instrucțiuni, astfel:

```

todays_date = get_date(todays_date);    show_date(get_date(todays_date));
show_date(todays_date);

```

Funcția precedentă *get\_date* primea o referință ca parametru și apoi returna aceeași referință. Așa cum veți învăța, mulți dintre manipulatorii de stream I/O au fost concepuți pentru o comportare asemănătoare. Manipulatorii primesc o referință ca parametru al stream-ului, realizează o operație, apoi returnează acea referință.

### FOLOSIREA REFERINTELOR LA OBIECTE

Dacă examinați diverse programe în C++, veți întâlni funcții care folosesc sau returnează referințe la obiecte. De exemplu, în Capitolul 11, veți învăța cum se creează propriii manipulatori de stream-uri I/O, cum sunt *hex*, *dec* sau *oct*. Veți defini atunci funcții manipulator ce lucrează cu stream-uri I/O similare cu cele arătate în programul următor, DECOCTHE.CPP:

```

#include <iostream.h>
#include <iomanip.h>

ios& dec(ios& stream) // Returnează o referință
{
    // Parametru tip referință
}

```

```

    stream.setf(ios::dec);
    return(stream);
}

ios& oct(ios& stream)
{
    stream.setf(ios::oct);
    return(stream);
}

ios& hex(ios& stream)
{
    stream.setf(ios::hex);
    return(stream);
}

void main(void)
{
    cout << oct << "Value 10 in octal is " << 10 << endl;
    cout << dec << "Value 10 in decimal is " << 10 << endl;
    cout << hex << "Value 10 in hexadecimal is " << 10 << endl;
}

```

Când manipulatorul apare într-o operație de ieșire, C++ va apela funcția manipulator corespunzătoare. După cum se vede, funcția primește ca parametru o referință la un stream I/O și apoi returnează o referință la același stream:

```

ios& dec(ios& stream)
{
    stream.setf(ios::dec);
    return(stream);
}

```

Următoarea funcție *tab*, de exemplu, inserează caracterul de tabulare într-un stream de ieșire:

```

ostream& tab(ostream& stream)
{
    stream << '\t';
    return(stream);
}

```

După cum se vede, manipulatorul inserează caracterul *tab* în stream-ul de ieșire specificat. Când operația s-a încheiat, funcția va returna referința la stream. Pentru a folosi manipulatorul, programul îl va include în stream-ul de ieșire, cum se arată mai jos:

```

cout << tab << "Indenting the program output" << endl;

```

## ATENȚIE LA OBIECTELE ASCUNSE

Când un program folosește referințe, trebuie să vă asigurați că tipul referinței corespunde cu cel al variabilei asociate. De exemplu, dacă se creează un alias la o variabilă de tipul *int*, trebuie să declarați referința de tip *int*:

```
cout << tab << "Indenting the program output" << endl;
```

```
int& alias = value;
```

Dacă tipul referinței nu concordă cu tipul variabilei, compilatorul va crea un obiect ascuns, căruia i se va atribui valoarea variabilei. De exemplu, următorul program, *WRONGREF.CPP*, creează o referință denumită *wrong\_type* de tipul *long* la care atribuie un alias la o variabilă de tipul *int*:

```
#include <iostream.h>
```

```
void main(void)
```

```
{
```

```
    int value = 1;
```

```
    long& wrong_type = value;
```

```
    cout << "Starting values: " << value << " and " << wrong_type << endl;
```

```
    value++;
```

```
    cout << "Ending values: " << value << " and " << wrong_type << endl;
```

```
}
```

La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> WRONGREF <Enter>
```

```
Starting values: 1 and 1
```

```
Ending values: 2 and 1
```

După cum se poate vedea, variabila și referința sa conțin valori diferite! Când compilatorul C++ a întâlnit referința la tipul *long*, el a alocat memorie pentru o altă mărime, căreia i-a asociat referința *wrong\_type*. După ce compilatorul a alocat o locație de memorie, el i-a atribuit acesteia valoarea variabilei pentru care a fost creat aliasul (adică valoarea 1). Exersați cu acest program, efectuând operații pe variabila *value* și pe referința *wrong\_type*. Veți vedea că variabila și referința sunt independente.

Majoritatea programelor nu vor folosi niciodată obiecte ascunse. Din păcate, acestea pot duce la erori greu de depanat. Prin urmare, trebuie să fiți conștient de această caracteristică a compilatorului.

## REZUMAT

La examinarea programelor C++, puteți întâlni funcții ce folosesc în mare măsură referințele. În cel mai simplu sens, o referință este un alias sau un al doilea nume folosit pentru o variabilă. În Capitolul 11, de exemplu, veți crea funcții manipulator ce primesc ca parametru o referință la un stream I/O și returnează apoi tot o referință la stream. Înainte de a continua cu Capitolul 11, asigurați-vă că ați învățat următoarele:

- ✓ O referință este un alias sau un al doilea nume folosit de program pentru a menționa o variabilă.
- ✓ Folosind unul sau mai multe nume asociate pentru aceeași variabilă, programele pot deveni mai greu de înțeles, prin creșterea numărului de identificatori pe care programatorul trebuie să-i rețină.
- ✓ Pentru a crea o referință, trebuie specificat un nume și un tip, plasând simbolul ampersand (&) imediat după numele tipului. În plus, e necesar ca referinței să i se atribuie o valoare inițială.
- ✓ O referință nu este o variabilă. De aceea, C++ restrânge setul de operații ce pot fi efectuate cu o referință.
- ✓ Pentru compilatorul C++, o referință este o adresă. Pentru program, o referință este văzută ca o valoare conținută la o adresă rezolvată.
- ✓ Prin folosirea referințelor, se pot elimina multe din operațiile cu pointeri. De aceea, multe programe C++ folosesc referințele pentru a transfera sau a returna structuri și obiecte către sau dinspre o funcție.
- ✓ Orice operație efectuată cu o referință poate fi efectuată și cu pointeri.
- ✓ Dacă tipul referinței nu coincide cu tipul variabilei asociate, compilatorul C++ va crea un obiect ascuns. Astfel de obiecte sunt dificil de detectat.

# S U C C E S C U +

## SECȚIUNEA II

### APROFUNDAREA

- 11 Aprofundarea stream-urilor din C++
- 12 Aprofundarea stream-urilor și
- 13 Aprofundarea funcțiilor virtuale
- 14 Aprofundarea tratării excepțiilor
- 15 Aprofundarea gestionării zonei de memorie alocabilă



## CAPITOLUL 11

### APROFUNDAREA STREAM-URILOR I/O DIN C++

În Capitolul 1 ați examinat stream-urile I/O din C++ și modul lor de folosire. Atunci ați învățat să folosiți diferiți manipulatori și funcții membru. Acest capitol analizează mai în amănunt clasele care formează stream-urile I/O din C++. Veți examina în detaliu clasele de bază și clasele derivate. În momentul încheierii acestui capitol veți putea înțelege pe deplin fișierele antet, cum sunt `IOSTREAM.H` și `IOMANIP.H`. În plus, veți învăța să creați propriii dumneavoastră manipulatori. În sfârșit, acest capitol examinează stream-ul de date ce are loc în fundal când este executată o operație de intrare/ieșire:

La terminarea acestui capitol, veți cunoaște următoarele:

- ♦ Cum se pot crea manipulatori cu parametri
- ♦ Cum se poate simplifica folosirea unui manipulator în alte programe
- ♦ Modul în care se pot asocia două stream-uri și efectul acestei operații
- ♦ Unde sunt definite stream-urile de fișier în C++
- ♦ Cum se pot realiza operații I/O formate
- ♦ Cum se realizează operații I/O neformate

**Observație:** *Compilerul pe care îl aveți la dispoziție poate folosi și alte fișiere antet, decât cele descrise în acest capitol. De exemplu, compilerul Borland C++ versiunea 4.0 folosește fișierul `IOSTREAM.H` pentru toate definițiile legate de stream-uri. În plus, față de `IOSTREAM.H`, compilerul Microsoft Visual C++ versiunea 1.0 folosește și fișierele `IOS.H`, `ISTREAM.H` și `OSTREAM.H`. Acest capitol folosește Borland C++ ca punct de referință; dacă folosiți alte compilatoare, va trebui să utilizați fișierele antet impuse de acestea.*

#### SĂ ÎNȚELEGEM RELAȚIA ÎNTRE CLASELE STREAM-URILOR I/O

Așa cum ați învățat, fișierul antet `IOSTREAM.H` definește clasele pentru stream-uri I/O. Dacă nu ați făcut-o încă, tipăriți acum o copie a acestui fișier. Când examinați conținutul acestui fișier, veți găsi clase de genul `ios`, `istream` și `ostream`. După cum se vede în Figura 11.1, clasa `ios` este clasa de bază, din care se derivează clasele `istream` și `ostream`. Clasele `istream` și `ostream` adaugă la clasa de bază `ios` operatorii de extracție (`>>`), respectiv de inserție (`<<`).

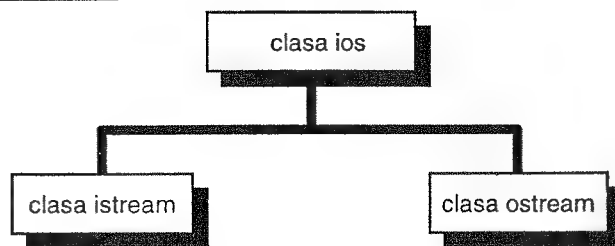


Figura 11.1 Clasele *istream* și *ostream* sunt derivate din clasa de bază *ios*.

Clasele *istream* și *ostream* implementează operatorii de extracție și inserție. Dacă examinați fișierul antet `IOSTREAM.H`, veți găsi prototipuri pentru operatorii de inserție și extracție, de forma:

```

istream_FAR & _Cdecl operator>> (istream_FAR & (_Cdecl *f)(istream_FAR
&));
istream_FAR & _Cdecl operator>> (ios_FAR & (_Cdecl *f)(ios_FAR & ));
istream_FAR & _Cdecl operator>> ( signed char_FAR *);
: : : : :
istream_FAR & _Cdecl operator>> (unsigned char_FAR *);

ostream_FAR & _Cdecl operator<< (short);
ostream_FAR & _Cdecl operator<< (unsigned short);
ostream_FAR & _Cdecl operator<< (int);
: : : : :
ostream_FAR & _Cdecl operator<< (unsigned int);
    
```

După cum se vede, fișierul furnizează operatorii pentru tipuri de date obișnuite, ca *short*, *unsigned short* sau *int*. În Capitolul 1 s-au folosit extensiv stream-urile I/O *cin* și *cout* pentru a realiza operații cu tastatura și ecranul. În plus, în Capitolul 1 s-au discutat pe scurt și stream-urile *cerr* și *clog*, ce facilitau afișarea unor mesaje pe dispozitivul de eroare standard. Dacă examinați fișierul antet `IOSTREAM.H`, veți găsi declarații similare celor ce urmează:

```

extern istream_withassign _Cdecl cin;
extern ostream_withassign _Cdecl cout;
extern ostream_withassign _Cdecl cerr;
extern ostream_withassign _Cdecl clog;
    
```

Se poate vedea că stream-ul de intrare *cin* este definit folosind clasa *istream\_withassign*. De asemenea, *cout*, *cerr* și *clog* sunt definiți folosind clasa stream de ieșire *ostream\_withassign*. Cum se arată în Figura 11.2, cele două tipuri de clase sunt de fapt derivate din clasele *ostream* și *istream*.

În general, obiectele clasei *withassign* acceptă operatorul de atribuire (=) care poate fi folosit pentru a atribui un stream altui stream sursă sau destinație. De

exemplu, stream-urile *cin* și *cout* sunt asociate în mod normal cu tastatura și ecranul. Dacă redirecționați intrarea și ieșirea programului, atunci conexiunea stream-urilor se va schimba.

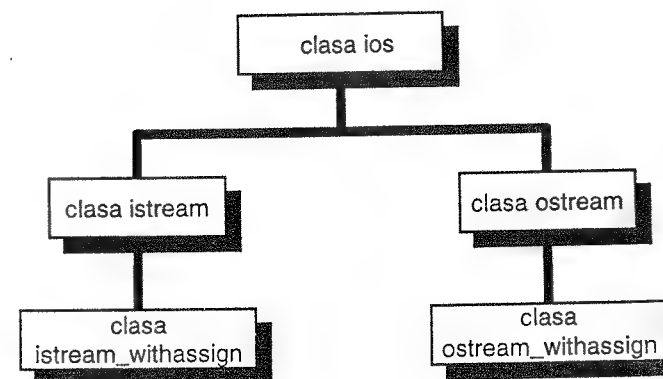


Figura 11.2 Tipurile de clasă *ostream\_withassign* și *istream\_withassign* sunt derivate din *ostream* și *istream*.

Următorul program, `CHSTREAM.CPP`, folosește operatorul de atribuire cu stream-ul *cout*, pentru a scrie ieșirea acestuia în fișierul `COUT.DAT` ce este în mod curent conectat la un stream fișier.

```

#include <fstream.h>

void main(void)
{
    ofstream output("COUT.DAT");

    cout << "About to assign cout to the file" << endl;

    cout = output;

    cout << "This is not written to the screen, but rather, the file!"
        << endl;
}
    
```

La compilarea și execuția programului, pe ecran va apărea primul mesaj. Al doilea mesaj nu va apărea pe ecran, ci va fi scris în fișierul `COUT.DAT`.

În mod analog, următorul program, `COUT_PRN.CPP`, atribuie stream-ul *cout* unui fișier ce corespunde dispozitivului de tipărire. La compilarea și execuția programului, acesta își va scrie ieșirea la dispozitivul de tipărire:

```
#include <fstream.h>

void main(void)
{
    ofstream cprn("PRN");

    cout << "About to assign cout to the printer" << endl;

    cout = cprn;

    cout << "This is not written to the screen, but rather, the printer!"
        << endl;
}
```



### SĂ ÎNȚELEM STREAM-URILE I/O

Un *stream* este cel mai bine definit ca o serie de octeți ce se deplasează de la o sursă spre o destinație. Fișierul antet `IOSTREAM.H` definește în C++ clasele de *stream* I/O. Aceste clase sunt bazate pe clasa *ios* care furnizează variabilele și funcțiile membru necesare în majoritatea operațiilor de intrare/ieșire. Fișierul antet derivează apoi clasele *istream* și *ostream* care, la rândul lor, implementează operatorii de extracție (>>) și inserție (<<) folosiți pentru operațiile de intrare/ieșire. Majoritatea programelor C++ folosesc *stream*-urile I/O *cin*, *cout*, *cerr* și *clog*. Acest *stream*-uri sunt definite de clasele *istream\_withassign* sau *ostream\_withassign*, ceea ce indică faptul că obiectele de acest tip pot fi conectate cu alte obiecte de același tip folosind funcția membru operator de atribuire (=)

```
extern istream_withassign _Cdecl cin;
extern ostream_withassign _Cdecl cout;
```

### CREAREA PROPRIILOR MANIPULATORI I/O

Așa cum ați învățat, un manipulator este o funcție ce poate fi plasată în interiorul unei operații de inserție sau extracție. În Capitolul 1 ați folosit manipulatori definiți în fișierul antet `IOMANIP.H`. Așa cum se va vedea, puteți crea proprii manipulatori pentru a-i folosi în *stream*-uri de intrare/ieșire. De exemplu, următoarea instrucțiune folosește un manipulator denumit *beep* ce activează difuzorul încorporat al calculatorului:

```
cout << beep << "Important message" << endl;
```

Următoarele instrucțiuni creează manipulatorul *beep*:

```
ostream& beep(ostream& stream)
{
    stream << '\007';
    return(stream);
}
```

După cum se vede, funcția returnează un obiect de tip *ostream*, de fapt același *stream* de ieșire pe care îl primește funcția drept parametru *stream*. Într-una din instrucțiunile funcției se inserează în *stream* caracterul ASCII de valoare 7. Funcția returnează apoi o referință la *stream*-ul actualizat. Când programul întâlnește un manipulator într-un operator de inserție sau de extracție, programul va apela funcția ce corespunde acestui manipulator. În instrucțiunea următoare, de exemplu, programul apelează funcția *beep* de două ori:

```
cout << beep << "Important message" << beep << endl;
```

Următorul program, `BEEPMAN.CPP`, folosește manipulatorul pentru activarea difuzorului înainte de afișarea unui mesaj:

```
#include <iostream.h>

ostream& beep(ostream& stream)
{
    stream << '\007';
    return(stream);
}

void main(void)
{
    cout << beep << "This is an important message" << endl;
}
```



### CREAREA PROPRIILOR MANIPULATORI

În această carte s-au folosit manipulatori I/O prezentați în fișierul antet `IOMANIP.H`. În afara acestor manipulatori, programul își poate crea proprii manipulatori. Pentru a crea un manipulator, se declară o funcție care returnează o referință la un obiect de tip *istream* sau *ostream*. După cum se vede în continuare, primul parametru al funcției corespunde *stream*-ului:

```

    Tip returnat
    Nume manipulator
    ostream& manipulator_name(ostream& stream_name)
    Parametru stream
```

În interiorul manipulatorului, funcția realizează prelucrarea dorită apoi returnează stream-ul actualizat:

```
{
    // Desired statements

    // Return the updated stream
    return(stream_name);
}
```

Parametru stream returnat

Următorul program, TAB\_MAN.CPP, folosește un manipulator denumit *tab*, care va insera un caracter de tabulare în stream-ul de ieșire:

```
#include <iostream.h>

ostream& tab(ostream& stream)
{
    stream << '\t';
    return(stream);
}

void main(void)
{
    cout << "Hello" << tab << "world!" << endl;
}
```

În Capitolul 1 ați învățat să folosiți indicatorii *ios::scientific* și *ios::fixed* pentru a selecta formatul de afișare a unei valori reale. De exemplu, următorul program, FIXEDSCI.CPP, folosește acești indicatori pentru a afișa valoarea 123.456:

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
{
    float value = 123.456;

    cout << setiosflags(ios::fixed) << value << endl;
    cout << setiosflags(ios::scientific) << value << endl;
}
```

La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> FIXEDSCI <ENTER>
123.456001
1.234560e+02
```

Dacă acești indicatori se folosesc frecvent, atunci este oportună crearea manipulatorilor *fixed* și *scientific*, ca mai jos:

```
ostream& fixed(ostream& stream)
{
    stream << setiosflags(ios::fixed);
    return(stream);
}

ostream& scientific(ostream& stream)
{
    stream << setiosflags(ios::scientific);
    return(stream);
}
```

Următorul program, FIXSCI\_2.CPP, folosește manipulatorii *fixed* și *scientific* pentru afișarea valorii 123.456:

```
#include <iostream.h>
#include <iomanip.h>

ostream& fixed(ostream& stream)
{
    stream << setiosflags(ios::fixed);
    return(stream);
}

ostream& scientific(ostream& stream)
{
    stream << setiosflags(ios::scientific);
    return(stream);
}

void main(void)
{
    float value = 123.456;

    cout << fixed << value << endl;
    cout << scientific << value << endl;
}
```

Următorul program, DECOCTHE.CPP, creează și folosește manipulatorii deja familiari, *dec*, *oct* și *hex*:

```
#include <iostream.h>
#include <iomanip.h>

ios& dec(ios& stream)
{
    stream.setf(ios::dec);
    return(stream);
}
```

```

ios& oct(ios& stream)
{
    stream.setf(ios::oct);
    return(stream);
}

ios& hex(ios& stream)
{
    stream.setf(ios::hex);
    return(stream);
}

void main(void)
{
    cout << oct << "Value 10 in octal is " << 10 << endl;
    cout << hex << "Value 10 in decimal is " << 10 << endl;
    cout << dec << "Value 10 in hexadecimal is " << 10 << endl;
}

```

Următorul program, SHOWBASE.CPP, creează manipulatorul *showbase*, care va dirija stream-ul să insereze Ox în fața valorilor în hexazecimal și O în fața celor scrise în octal:

```

#include <iostream.h>
#include <iomanip.h>

ios& showbase(ios& stream)
{
    stream.setf(ios::showbase);
    return(stream);
}

void main(void)
{
    cout << showbase;
    cout << oct << "Value 10 in octal is " << 10 << endl;
    cout << hex << "Value 10 in decimal is " << 10 << endl;
    cout << dec << "Value 10 in hexadecimal is " << 10 << endl;
}

```

La compilarea și execuția acestui program, pe ecran va apărea:

```

C:\> SHOWBASE <ENTER>
Value 10 in octal is 012
Value 10 in decimal is 0xa
Value 10 in hexadecimal is 10

```

În sfârșit, următorul program, SHOW\_PO.CPP, creează manipulatorii *showpoint* și *showpos* care dirijează stream-ul pentru a afișa punctul la valorile reale, respectiv pentru a preceda numerele pozitive cu semnul plus:

```

#include <iostream.h>
#include <iomanip.h>

ios& showpoint(ios& stream)
{
    stream.setf(ios::showpoint | ios::fixed);
    return(stream);
}

ios& showpos(ios& stream)
{
    stream.setf(ios::showpos);
    return(stream);
}

void main(void)
{
    cout << 10.0 / 2 << endl;
    cout << showpoint;
    cout << 10.0 / 2 << endl;

    cout << 1 << ' ' << 2 << ' ' << endl;
    cout << showpos;
    cout << 1 << ' ' << 2 << ' ' << endl;
}

```

La compilarea și execuția acestui program, pe ecran va apărea:

```

C:\> SHOW_PO <ENTER>
5
5.000000
1 2
+1 +2

```

### MANIPULATORI CU PARAMETRI MULTIPLI

Fiecare din manipulatorii anteriori a fost destul de simplu, prin aceea că nu acceptă parametri. Următorul manipulator, denumit *spaces*, permite inserarea unui anumit număr de spații într-un stream de ieșire:

```
cout << spaces(5) << "Just skipped 5 spaces" << endl;
```

Pentru a crea un manipulator cu parametri, trebuie create două funcții. Prima funcție este foarte asemănătoare celor create până acum:

```

ostream& spaces(ostream& stream, int space_count)
{
    for (int i = 0; i < space_count; i++)

```



```

    stream << ' ';

    return(stream);
}

```

Se observă că funcția returnează o referință la un stream de ieșire și primește tot o referință drept parametru. În plus, funcția primește al doilea parametru de tip *int*, denumit *space\_cout*. A doua funcție folosește macroinstrucțiunea OMANIP, care este definită în fișierul antet IOMANIP.H, cum se arată mai jos:

```

OMANIP(int) spaces(int space_count)
{
    OMANIP(int) type(spaces, space_count);
    return(type);
}

```

În timpul compilării, preprocesorul va expanda macroinstrucțiunea, pentru a crea funcția apelată de program la apariția manipulatorului. După cum se observă, această funcție apelează prima funcție, specificând cei doi parametri.

Următorul program, SPACEMAN.CPP, folosește manipulatorul *spaces* pentru a insera un număr variabil de spații în stream-ul de ieșire:

```

#include <iostream.h>
#include <iomanip.h>

ostream& spaces(ostream& stream, int space_count)
{
    for (int i = 0; i < space_count; i++)
        stream << ' ';

    return(stream);
}

OMANIP(int) spaces(int space_count)
{
    OMANIP(int) type(spaces, space_count);
    return(type);
}

void main(void)
{
    for (int i = 1; i <= 5; i++)
        cout << "Skipping " << i << spaces(i) << "spaces" << endl;
}

```

**Observație:** Este posibil ca unele compilatoare să nu accepte total macroinstrucțiunea OMANIP. Dacă obțineți erori de sintaxă la compilarea programului, apăsați la un manual de referință al compilatorului.

La compilarea și execuția programului, pe ecran va apărea:

```

C:\> SPACEMAN <ENTER>
Skipping 1 spaces
Skipping 2 spaces
Skipping 3 spaces
Skipping 4 spaces
Skipping 5 spaces

```

În același mod, următorul program, NEWLINES.CPP, creează un manipulator denumit *newlines*, care dirijează stream-ul să tipărească un anumit număr de linii albe:

```

#include <iostream.h>
#include <iomanip.h>

ostream& newlines(ostream& stream, int count)
{
    for (int i = 0; i < count; i++)
        stream << endl;

    return(stream);
}

OMANIP(int) newlines(int count)
{
    OMANIP(int) type(newlines, count);
    return(type);
}

void main(void)
{
    for (int i = 1; i <= 5; i++)
        cout << "Skipping " << i << newlines(i) << "lines" << endl;
}

```

**Observație:** Este posibil ca unele compilatoare să nu accepte total macroinstrucțiunea OMANIP. În caz de erori sintactice la compilarea programului, apăsați la documentația compilatorului.

#### CHEIA SUCCESULUI



#### MANIPULATORI CU PARAMETRI

La crearea propriilor manipulatori, este posibil de a transfera acestora o anumită valoare. Pentru a crea un manipulator cu parametru, trebuie, de fapt, create două funcții. Prima funcție specifică numele manipulatorului, parametrii și instrucțiunile. De exemplu, următorul manipulator, *beeps*, specifică de câte ori se activează clopoțelul calculatorului:

```
ostream& bells(ostream& stream, int bell_count)
{
    for (int i = 0; i < bell_count; i++)
        stream << '\a';

    return(stream);
}
```

A doua funcție folosește macrocomanda *OMANIP* definită în fișierul antet *IOMANIP.H*:

```
OMANIP(int) bells(int bell_count)
{
    OMANIP(int) type(bells, bell_count);
    return(type);
}
```

Pentru a activa clopoțelul de 3 ori, de exemplu, se va folosi manipulatorul astfel:

```
cout << bells(3);
```

În sfârșit, următorul program, *SETFILL.CPP*, creează un manipulator *setfill* de forma celor discutate în Capitolul 1:

```
#include <iostream.h>
#include <iomanip.h>

ostream& filler(ostream& stream, int fillchar)
{
    stream.fill((char)fillchar);
    return(stream);
}

OMANIP(int) filler(char fillchar)
{
    OMANIP(int) type(filler, fillchar);
    return(type);
}

void main(void)
{
    cout << "Hello" << filler('.') << setw(15) << "world!" << endl;
}
```

**Observații:** Unele compilatoare nu acceptă macroinstrucțiunea *OMANIP*. În caz de erori sintactice, apelați la documentația compilatorului.

La compilarea și execuția programului, pe ecran va apărea:

```
C:\> SETFILL <ENTER>
Hello.....world!
```

Folosind această tehnică se poate crea o mare varietate de manipulatori. Încercați, de exemplu, să creați manipulatori care aliniază la stânga sau la dreapta ieșirea unui program.

### CREAREA UNUI FIȘIER DE MANIPULATORI PROPRII

Așa cum ați învățat, fișierul antet *IOMANIP.H* conține prototipul funcțiilor câtorva manipulatori obișnuiți. În funcție de numărul și complexitatea manipulatorilor folosiți într-un program, s-ar putea dori plasarea acestor manipulatori (cu definițiile complete) într-un fișier antet specific, cum ar fi *MY\_MANIP.H*. Dacă nu doriți ca și alți utilizatori să aibă acces la manipulatori, plasați prototipul acestora în fișierul antet și codul efectiv al manipulatorilor într-o bibliotecă de obiecte. În Capitolul 17 găsiți pașii care trebuie urmați pentru a crea o bibliotecă de clase. Aceiași pași trebuie realizați și în cazul creării unei biblioteci de manipulatori proprii.

### SĂ ÎNȚELEM STREAM-URILE I/O ASOCIATE

În Capitolul 1 ați învățat că stream-urile I/O *cout* și *clog* realizează o ieșire prin buffer. Prin urmare, ieșirea scrisă în aceste stream-uri nu apare pe ecran până când nu este îndeplinită una din condițiile: bufferul este plin, programul se termină, programul golește bufferul, sau, în cazul lui *cout*, programul realizează o operație de intrare de la *cin*. În acest caz, se spune că *cin* este asociat cu stream-ul de ieșire *cout*. Dacă examinați fișierul antet *IOSTREAM.H*, veți afla că clasa *ios* acceptă o funcție membru denumită *tie*. Dacă programul apelează funcția *tie* fără parametru, *tie* va returna un pointer la stream-ul de ieșire la care este asociat stream-ul de intrare. Următorul program, *SHOWTIE.CPP*, folosește funcția *tie* pentru a verifica dacă stream-ul *cin* este asociat cu stream-ul *cout*:

```
#include <iostream.h>

void main(void)
{
    if (*(cin.tie()) == cout)
        cout << "cin is tied to cout" << endl;
    else
        cout << "cin not tied to cout" << endl;
}
```

Funcția membru *tie* permite, de asemenea, asocierea dintre un stream de intrare și unul de ieșire. Următorul program, *TIE\_CON.CPP*, deschide un fișier ce corespunde dispozitivului consolă. Apoi, programul asociază stream-ul *cin* cu stream-ul fișier. După aceasta se scrie un mesaj în stream și se așteaptă 3 secunde. Când programul realizează operația de intrare pe *cin*, mesajul va fi afișat fără întârziere pe *cout*:

```
#include <fstream.h>
#include <time.h>

void main(void)
{
    time_t start_time, current_time;

    ofstream screen("CON");

    cin.tie(&screen);

    screen << "Hello C++ world!--Press Enter to continue";
    time(&start_time);

    do {
        time(&current_time);
    } while ((current_time - start_time) < 3);

    cin.get();
    screen.close();
}
```

Se poate, de asemenea, asocia un stream de ieșire cu un alt stream de ieșire. În acest mod, când bufferul de ieșire este golit, la fel va fi și bufferul de ieșire al stream-ului asociat. În mod prestabilit, C++ asociază *cerr* la *cout* și *clog* la *cout*. Dacă *cerr* sau *clog* este golit, la fel va fi și *cout*. Următorul program, SHOWTIES.CPP, arată cum sunt asociate stream-urile în mod prestabilit în C++:

```
#include <iostream.h>

void main(void)
{
    if (*(cin.tie()) == cout)
        cout << "cin is tied to cout" << endl;

    if (*(cerr.tie()) == cout)
        cout << "cerr is tied to cout" << endl;

    if (*(clog.tie()) == cout)
        cout << "clog is tied to cout" << endl;

    if (*(cout.tie()) == NULL)
        cout << "cout is not tied" << endl;
}
```

La compilarea și execuția acestui program, ecranul va afișa:

```
C:\> SHOWTIES <ENTER>
cin is tied to cout
cerr is tied to cout
```

clog is tied to cout  
cout is not tied

## MAI MULTE DESPRE INTERACȚIUNEA CLASELOR DE STREAM-URI

În Capitolul 3 ați examinat stream-urile fișier în C++. Așa cum ați învățat, fișierul *antet* FSTREAM.H definește clasele pentru stream-uri fișier *ifstream*, *ofstream* și *fstream*. Când în programe trebuie să se realizeze operații de intrare pe fișiere, atunci se creează obiecte de tipul *ifstream*. Când se execută operații de ieșire, se folosesc obiecte de tipul *ofstream*. În sfârșit, când același fișier trebuie citit și scris, programele folosesc obiecte de tipul *fstream*. Dacă examinați declarațiile acestor clase din fișierul FSTREAM.H, veți vedea că aceste clase se bazează pe clasele *istream*, *ostream* și *iostream*, cum se arată în Figura 11.3.

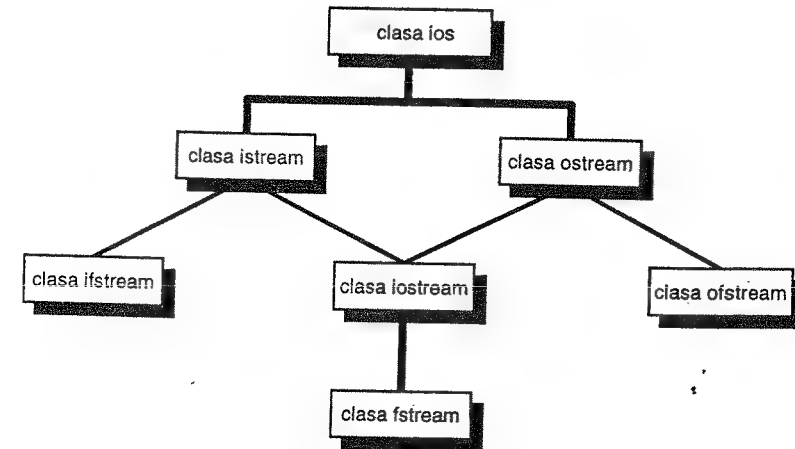


Figura 11.3. Clasele *ifstream*, *ofstream* și *fstream* sunt derivate din clasele *istream*, *ostream* și *iostream*.

Clasele *ifstream* și *ofstream* adaugă la clasele de bază din care provin funcțiile membru *open* și *close*. Clasa *iostream* adaugă facilitatea de citire și scriere la clasele moștenite *istream* și *ostream*. Clasa *fstream* adaugă funcțiile *open* și *close* la cele furnizate deja de clasa *iostream*.

## SĂ ÎNȚELEM IEȘIREA FORMATATĂ ȘI NEFORMATATĂ

Când realizați operații I/O, se pot folosi clase pentru intrări/ieșiri formate sau neformate. Toate clasele discutate în acest capitol au realizat I/O formatat. Tabela 11.1 prezintă clasele pentru I/O formate.

Numele clasei	Descriere
ios	Clasa de bază pentru stream-uri de intrare și ieșire
istream	Derivată din <i>ios</i> , adaugă operatorul de extracție
ostream	Derivată din <i>ios</i> , adaugă operatorul de inserție
iostream	Derivată din <i>istream</i> și <i>ostream</i> , adaugă operațiile de citire și scriere
istream_withassign	Derivată din <i>istream</i> , adaugă operatorul de atribuire
ostream_withassign	Derivată din <i>ostream</i> , adaugă operatorul de atribuire
ifstream	Derivată din <i>istream</i> , adaugă metodele de închidere și deschidere
ofstream	Derivată din <i>ostream</i> , adaugă metodele de închidere și deschidere
fstream	Derivată din <i>iostream</i> , adaugă metodele de închidere și deschidere
stdiostream	Derivată din <i>ios</i> , adaugă o conexiune la un fișier din <i>stdio</i>

Tabela 11.1. Clasele C++ pentru I/O formatat

Când programele realizează intrări/ieșiri formate, clasele I/O convertesc valorile întregi și reale în caractere ASCII corespunzătoare. De exemplu, următoarea instrucțiune tipărește valorile 22, 33 și 44:

```
cout << 22 << " " << 33 << " " << 44 << endl;
```

Pentru a afișa valoarea 22, de exemplu, programul tipărește valoarea ASCII 50 (valoarea pentru caracterul '2') de două ori. Analog, pentru a tipări valoarea 33, programul va tipări valoarea ASCII 51 de două ori.

Când programele realizează afișare *neformatată*, programul tipărește reprezentarea binară a fiecărei valori, fără conversii. Tabela 11.2 prezintă clasele C++ pentru I/O neformatat.

Numele clasei	Descriere
streambuf	Clasa de bază pentru I/O prin buffer care implementează pointerii de citire și scriere și operații I/O simple
filebuf	Derivată din <i>streambuf</i> , adaugă metodele de deschidere și închidere
stdiobuf	Derivată din <i>streambuf</i> , adaugă o conexiune la un fișier din <i>stdio</i>

Tabela 11.2 Clasele C++ pentru I/O neformatat

Pentru a înțelege mai bine ierarhia de clase pentru I/O neformatat, să privim diagrama din Figura 11.4.

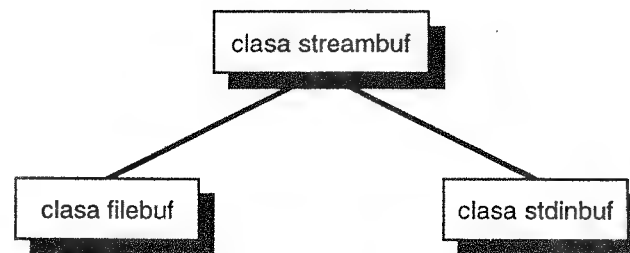


Figura 11.4. Clasele C++ pentru I/O neformatat.

Fișierul antet IOSTREAM.H definește clasele pentru I/O neformatat. Când programele realizează I/O neformatat, se lucrează în principiu cu date în buffer, cum se arată în continuare.

### SĂ ÎNȚELEGEM INTRĂRILE/IEȘIRILE PRIN BUFFER

Clasa *streambuf* oferă baza pentru operații I/O în C++ folosind buffer-ul. În asemenea situații, programele citesc sau scriu informații într-un anumit buffer (o zonă tampon de memorie), cum se arată în Figura 11.5.

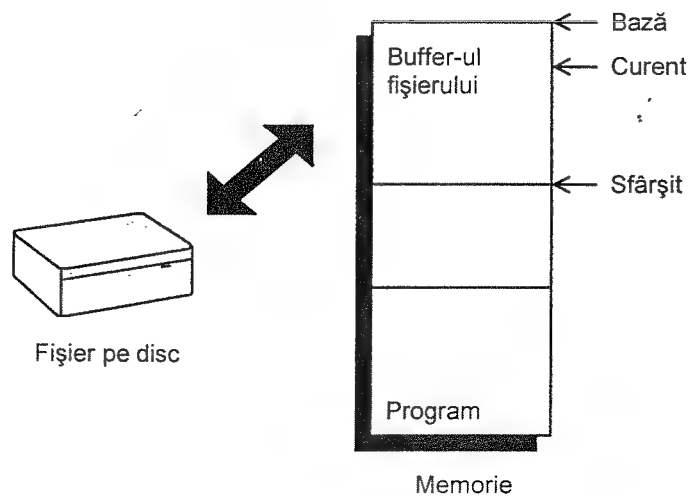


Figura 11.5. I/O prin buffer plasează datele în buffer-ul de memorie

După cum se observă, pentru controlul buffer-ului se folosesc câțiva pointeri. Dacă examinați fișierul antet IOSTREAM.H, veți vedea că în fișier sunt definite diferite funcții membru ale claselor. Multe din aceste funcții manipulează pointeri la buffere sau caractere referite de pointeri. Nu se pot crea obiecte de tipul *streambuf*. În schimb, se pot deschide fișiere folosind clasa *filebuf*. În majoritatea claselor, însă, programele nu vor realiza I/O prin buffer folosind obiecte de tipul *filebuf*, ci vor folosi obiecte de tipul *ifstream* sau *ofstream*, așa cum am văzut în Capitolul 3. Clasa *ios* folosește un pointer la un obiect de tipul *streambuf* pentru a putea realiza I/O prin buffer. Este important de reținut că clasa *ios* nu derivă din clasa *streambuf*, dar folosește, în schimb, un pointer la un obiect de tipul *streambuf*.

Pentru a manipula obiecte ale clasei *filebuf*, trebuie să înțelegem mai întâi scopul funcțiilor membru ale clasei *streambuf*. Dacă examinați fișierul IOSTREAM.H, veți găsi prototipurile unor funcții membru de forma următoare:

```
int _Cdecl sgetc();           // Peek at next character in
                              // the buffer
int _Cdecl sngetc();          // Advance to and return next
                              // character in the buffer
int _Cdecl sbumpc();           // Return the current
                              // character in the buffer
                              // and advance the pointer
void _Cdecl stosscc();         // Advance to the next
                              // character in the buffer
int _Cdecl sgetn(char _FAR *, int); // Get next n characters in
                              // the buffer
int _Cdecl sputbackc(char);    // Return a character to the
                              // buffer
int _Cdecl in_avail();         // Number of available
                              // characters in the buffer
int _Cdecl sputc(int);         // Put one character into the
                              // buffer
int _Cdecl sputn(const char _FAR *, int); // Put n characters into
                              // the buffer
int _Cdecl out_waiting();      // Number of unflushed
                              // characters in output
                              // buffer
```

Pe lângă funcțiile membru prezentate anterior, clasa *streambuf* definește funcții care permit căutarea sau deplasarea în buffer și funcții care sincronizează bufferul pentru operații de intrare/ieșire. La realizarea operațiilor prin buffer cu fișiere folosind obiecte de tipul *filebuf*, se folosesc una sau mai multe din aceste funcții. Pentru a le înțelege mai bine, Tabela 11.2 listează scopul funcțiilor și valoarea returnată de acestea.

## 11: Aprofundarea stream-urilor I/O

Nume funcție	Scop
sgetc	Anticipează următorul caracter din buffer. Dacă aceasta nu există, se returnează EOF. Pointerul <i>get</i> nu este deplasat.
sngetc	Deplasează pointerul <i>get</i> înainte cu un caracter și returnează următorul caracter.
sbumpc	Returnează caracterul curent apoi avansează pointerul <i>get</i> . Dacă caracterul nu există, se returnează EOF.
stosscc	Deplasează pointerul <i>get</i> înainte, cu un caracter.
sgetn	Returnează următoarele <i>n</i> caractere din buffer. Funcția întoarce numărul de caractere returnate. Dacă nu există caractere, se returnează EOF.
sputbackc	Plasează înapoi în buffer ultimul caracter luat din buffer.
in_avail	Returnează numărul de caractere disponibile.
sputc	Plasează caracterul specificat în buffer și avansează pointerul <i>put</i> .
sputn	Plasează <i>n</i> caractere dintr-un șir în buffer, avansând pointerul <i>put</i> .
out_waiting	Returnează numărul de caractere din bufferul de ieșire.

Tabela 11.3 Scopul funcțiilor membru *filebuf*.

### DESCHIDEREA UNUI FIȘIER PENTRU OPERAȚII I/O PRIN BUFFER

În majoritatea cazurilor, programele folosesc clasele *ifstream* și *ofstream* pentru operații I/O cu fișiere. Însă, prin examinarea operațiilor cu fișiere ce folosesc clasa *filebuf* veți înțelege mai bine operațiile I/O prin buffer. Deschiderea unui fișier pentru operații I/O prin buffer folosind clasa *filebuf* este foarte asemănătoare cu deschiderea unui stream fișier pentru operații de intrare/ieșire. Mai întâi, se declară un obiect de tipul *filebuf*, ca mai jos:

```
filebuf buffer_file;
```

Apoi, se deschide fișierul folosind funcția membru *open*, astfel:

```
buffer_file.open("FILENAME.EXT", ios::in);
```

Următorul program, OPENBUF.CPP, deschide un fișier specificat în linia de comandă pentru intrare prin buffer. Dacă fișierul va fi deschis cu succes, programul afișează un mesaj, iar apoi fișierul se închide. Dacă fișierul nu este deschis, programul afișează un mesaj de eroare corespunzător:

```
#include <fstream.h>
```

```
void main(int argc, char **argv)
```

```
{
    filebuf buffered_file;

    buffered_file.open(argv[1], ios::in);

    if (buffered_file.is_open())
    {
        cout << "File successfully opened" << endl;
        buffered_file.close();
    }
    else
        cerr << "Error opening the file: " << argv[1] << endl;
}
```

După cum se poate vedea, programul folosește metoda *close* pentru a închide fișierul când nu va mai fi necesar. Următorul program, SHOWBUF.CPP, deschide un fișier pentru intrare prin buffer. Programul afișează apoi conținutul fișierului, caracter cu caracter:

```
#include <fstream.h>

void main(int argc, char **argv)
{
    filebuf buffered_file;

    int letter;

    buffered_file.open(argv[1], ios::in);

    if (buffered_file.is_open())
    {
        while ((letter = buffered_file.sgetc()) != EOF)
        {
            cout.put((char)letter);
            buffered_file.snexct();
        }

        buffered_file.close();
    }
    else
        cerr << "Error opening the file: " << argv[1] << endl;
}
```

Programul folosește funcția membru *sgetc* pentru a anticipa următorul caracter din buffer. Dacă sfârșitul de fișier nu a fost întâlnit, programul afișează caracterul și apoi avansează pointerul la următorul caracter, folosind funcția *snexct*.

În mod similar, următorul program, EASYSHOW.CPP, deschide un fișier aflat în buffer. Programul afișează conținutul fișierului, folosind operatorul de inserție pentru a afișa bufferul fișierului. Dacă examinați fișierul antet IOSTREAM.H, veți

descoperi că operatorul de inserție este suprapus pentru operațiile care folosesc o adresă la *streambuf*. Funcția corespunzătoare acestui operator extrage caracterele din buffer, plasându-le în stream-ul de ieșire:

```
#include <fstream.h>

void main(int argc, char **argv)
{
    filebuf buffered_file;
    buffered_file.open(argv[1], ios::in);
    if (buffered_file.is_open())
    {
        cout << &buffered_file;
        buffered_file.close();
    }
    else
        cerr << "Error opening the file: " << argv[1] << endl;
}
```

## IMAGINE DE ANSAMBLU

Pentru a înțelege cum lucrează diferitele clase I/O, trebuie avut în vedere că clasele I/O conțin funcții publice ce pot fi apelate de programe și membri de tip *private* ce controlează operațiile I/O. La deschiderea unui fișier pentru intrare, de exemplu, este creat un obiect de tipul *ifstream*. Așa cum ați citit, clasa *ifstream* (care adaugă membrii *open* și *close*) este bazată pe clasa *istream*. Clasa *istream*, la rândul ei, este derivată din clasa de bază *ios*, adăugând la aceasta operatorul de extracție. Când programele realizează operații de intrare folosind stream-ul, datele trebuie plasate în anumite zone ale memoriei. Cum s-a arătat, datele sunt plasate într-un buffer de memorie ce corespunde unui obiect *streambuf*. Obiectul *streambuf* conține pointeri și funcții ce controlează accesul la date. Țineți minte, totuși, că clasa *ios* nu este derivată din clasa *streambuf*, ci folosește un pointer la un obiect *streambuf*.

## REZUMAT

Aproape toate programele C++ folosesc intens stream-uri I/O și stream-uri fișiere. Acest capitol a analizat în amănunt stream-urile I/O - atât modul lor de folosire, cât și relațiile dintre diferite tipuri de clase. Înainte de a trece la Capitolul 12, asigurați-vă că ați învățat următoarele:

- ✓ Fișierul antet IOSTREAM.H definește clasele de stream I/O. În cadrul fișierului, găsiți o serie de definiții de clase asociate.
- ✓ Clasa *ios* definește clasa de bază pentru operații I/O. Clasele *istream* și *ostream* adaugă la clasa *ios* operatorii de inserție și

extracție. În cadrul fișierului IOSTREAM.H, găsiți funcțiile membru ce definesc aceste operații.

- ✓ Stream-ul *cin* este definit ca *istream\_with\_assign*. Analog, stream-urile *cout*, *cerr* și *clog* sunt definite ca *ostream\_with\_assign*. Clasele definite ca *\_withassign* conțin operatorul de atribuire (=), ce poate fi folosit pentru a atribui un alt stream unui obiect al clasei.
- ✓ C++ implementează manipulatorii prin funcții, în același mod în care programele realizează suprapunerea operatorilor. Pentru a crea un manipulator pentru un stream I/O, se creează o funcție ce primește o referință la un stream drept parametru, apoi se manipulează stream-ul într-un anumit mod. Funcția returnează apoi referința la stream.
- ✓ Se pot crea și manipulatori cu parametri. Cel mai simplu mod de a crea un astfel de manipulator este de a folosi macroinstrucțiunea *OMANIP* definită în fișierul antet *IOMANIP.H*.
- ✓ Dacă se creează mai mulți manipulatori, aceștia se pot salva într-un fișier antet, simplificând folosirea lor în alte programe.
- ✓ Pentru un bun control al buffer-ilor I/O, programele pot asocia un stream cu alt stream. De exemplu, stream-ul *cin* este asociat cu *cout*. Dacă stream-ul *cout* conține ieșirea prin buffer care nu a fost încă afișată, atunci aceasta va apărea imediat ce se realizează o intrare de la *cin*.
- ✓ Fișierul antet *FSTREAM.H* definește stream-urile fișier în C++. În cadrul acestui fișier, găsiți definițiile claselor *ifstream*, *ofstream* și *fstream*. Aceste clase adaugă metodele *open* și *close* la cele moștenite din clasele *istream*, *ostream* și *iostream*.
- ✓ Când programele realizează operații I/O formate, informația de ieșire este mai întâi convertită în formatul ASCII. De asemenea, informația de intrare este convertită din ASCII în formatul dorit. Operațiile I/O neformate, pe de altă parte, nu convertesc valorile.
- ✓ Pentru a realiza operații I/O neformate, programele pot folosi obiecte *Filebuf*.

## CAPITOLUL 12

### APROFUNDAREA STREAM-URILOR ȘIR

În cuprinsul acestei cărți programele folosesc extensiv stream-urile I/O pentru a realiza operații cu tastatura, ecranul sau fișierele. C++ furnizează, de asemenea, un set de *stream-uri șir* ce pot fi folosite pentru păstrarea caracterelor ce urmează a fi afișate formatat sau pentru conversii numerice. De exemplu, dacă un șir conține reprezentarea ASCII a numărului 1001, se poate folosi un stream pentru a converti caracterele într-o valoare de tip *int*. Acest capitol examinează stream-urile șir în detaliu. Așa cum veți vedea, folosirea acestor stream-uri este asemănătoare multor operații efectuate cu stream-urile I/O. La terminarea acestui capitol, veți cunoaște:

- ◆ Ce este un stream șir și ce realizează
- ◆ Unde se găsesc definițiile claselor pentru stream-uri șir
- ◆ Care sunt cele mai importante clase pentru stream-uri șir
- ◆ Ce sunt stream-urile șir de intrare
- ◆ Ce sunt stream-urile șir de ieșire
- ◆ Cum acceptă stream-urile funcții membru
- ◆ Ce este un buffer dinamic

#### SĂ ANALIZĂM FIȘIERUL *STRSTREA.H*

C++ definește clasele de stream-uri șir în fișierul antet *STRSTREA.H*. Tipăriți acum o copie a conținutului fișierului. Puteți folosi conținutul fișierului ca o referință în cursul discuțiilor din acest capitol.

Pentru a folosi un stream șir, trebuie să includeți fișierul antet la începutul programului, cum se arată mai jos:

```
#include <strstrea.h>
```

Dacă examinați conținutul fișierului, veți vedea că se definesc o serie de clase. Cele mai cunoscute 3 clase de stream șir sunt descrise pe scurt în Tabela 12.1. În secțiunile următoare aceste clase vor fi descrise în detaliu:

Stream-ul	Scopul
istrstream	Suportă operații de formatare a șirurilor de intrare pentru șiruri membru
ostrstream	Suportă operații de formatare a șirurilor de ieșire pentru șiruri din memorie
strstream	Suportă operații de formatare a șirurilor de intrare sau ieșire pentru șiruri din memorie

Tabela 12.1. Clasele de stream-uri și definite în fișierul antet STRSTREA.H

## FOLOSIREA STREAM-URILOR ȘIR DE INTRARE

Cel mai bun mijloc de a înțelege stream-urile șir este de a examina câteva programe simple care folosesc aceste stream-uri. Stream-urile șir acceptă operații de intrare sau ieșire. În general, un stream șir de intrare acceptă operatorul de extracție (>>), permițând programelor să introducă variabile din buffer-ul stream-ului. Un stream șir de ieșire, pe de altă parte, acceptă operatorul de inserție (<<). Prin folosirea acestuia, programele pot scrie în stream caractere, șiruri, numere, și chiar caractere speciale. Stream-urile șir de intrare sunt adesea folosite pentru a converti o reprezentare ASCII a unui număr, la o reprezentare de tip întreg sau real. Stream-urile șir de intrare sunt adesea folosite pentru a converti o reprezentare a unui număr, la o valoare *int* sau *float*. De exemplu, să presupunem că șirul de caractere *number* conține valoarea 1.2345, cum se arată în Figura 12.1.



Figura 12.1. Reprezentarea ASCII a unei valori reale

Înainte de a lucra cu valoarea (de a executa operații matematice asupra acestuia), programul trebuie mai întâi să convertească valoarea ASCII în real. În limbajul C, s-ar putea folosi funcția de bibliotecă *atof*, așa cum se arată în următorul program, ATOF.C:

```
#include <stdlib.h>
#include <stdio.h>

void main(void)
{
    char *number = "1.2345";
    float value;
```

## 12: Aprofundarea stream-urilor și

```
value = atof(number);
```

```
printf("The value is %f\n", value);
printf("The value squared is %f\n", value * value);
}
```

La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> ATOF <ENTER>
The value is 1.234500
The value squared is 1.523990
```

În cadrul programelor C++ se pot încă folosi funcții ca *atof* pentru a realiza asemenea conversii.

Partea neplăcută a folosirii acestor funcții este că trebuie reținut numele funcțiilor (ca *atoi*, *atof*, *atol* etc.) De aceea, unii programatori C folosesc funcția *sscanf* pentru a citi valori dintr-un șir de caractere, eliminând necesitatea de a memora nume de funcții. Următorul program, SSCANF.C, citește o valoare reală dintr-un șir ASCII, realizând conversia dorită:

```
#include <stdio.h>

void main(void)
{
    char *number = "1.2345";
    float value;

    sscanf(number, "%f", &value);

    printf("The value is %f\n", value);
    printf("The value squared is %f\n", value * value);
}
```

Conversia valorilor folosind *sscanf* este foarte asemănătoare cu citirea valorilor dintr-un stream șir de intrare. Diferența, totuși, este aceea că la stream-urile șir nu trebuie folosit specificatorul de format, cum este %f. În locul acestuia se folosește operatorul de extracție (>>). Următorul program, INPUTSTR.CPP, ilustrează modul în care se folosește un stream șir pentru a converti o valoare reală:

```
#include <strstream.h>
#include <iostream.h>

void main(void)
{
    char *number = "1.2345";
    float value;

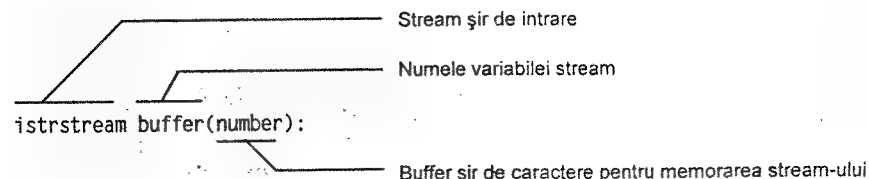
    istrstream buffer(number);
```



```
buffer >> value;

cout << "The value is " << value << endl;
cout << "The value squared is " << value * value << endl;
}
```

Programul creează un stream șir de intrare, denumit *buffer*, folosind următoarea instrucțiune:



La transmiterea unui singur buffer șir de caractere constructorului clasei *istream*, se presupune că lungimea buffer-ului șir (primul parametru) corespunde caracterului NULL de terminare a șirului. Următoarele instrucțiuni, pe de altă parte, creează un buffer de 256 octeți:

```
char °ir[256];

istream buffer(°ir, sizeof(°ir));
```

Dacă se specifică lungimea buffer-ului în timpul creării obiectului, stream-ul va accepta numărul de caractere specificat.

**SĂ ÎNȚELEM STREAM-URILE ȘIR DE INTRARE**

Un stream șir este un buffer de memorie ce conține caractere. Stream-urile șir de intrare sunt denumite astfel deoarece programele pot folosi operatorul de extracție (>>) pentru a atribui valori din aceste stream-uri unor variabile. Pentru a crea un stream de intrare, trebuie declarat un obiect de tipul *istream*, ca mai jos:

```
char °ir[256];
char *book = "Success with C++";

istream buffer(°ir, sizeof(°ir));
istream text(book);
```

Prima declarație de stream (*buffer*) specifică dimensiunea buffer-ului corespunzător stream-ului. În cazul celei de-a doua declarație (*text*), dimensiunea buffer-ului corespunde terminatorului NULL de la sfârșitul șirului „Succes cu C++”. Folosind operatorul de extracție, programele pot atribui date din aceste stream-uri unor variabile, cum se arată mai jos:

```
buffer >> value;
text >> word;
```

Stream-urile șir de intrare sunt frecvent folosite pentru a converti numere în reprezentarea ASCII în echivalentele lor întregi sau reale.

### CITIREA MAI MULTOR VALORI DINTR-UN STREAM ȘIR DE INTRARE

La citirea datelor din stream-uri de intrare, cum sunt *cin* sau stream-uri fișier, se pot citi mai multe valori la un moment dat folosind operatorul de extracție:

```
cout << "Type your first and last name: ";
```

```
cin >> first >> last;
```

Când se lucrează cu stream-uri șir de intrare, este posibil ca buffer-ul șirului să conțină mai multe valori. De exemplu, buffer-ul șir ar putea conține valorile 1, 100 și 1001, cum se arată în Figura 12.2

1		1	0	0		1	0	0	1
---	--	---	---	---	--	---	---	---	---

Figura 12.2 Valori întregi multiple într-un buffer de stream șir

Folosind operatorul de extracție, se pot citi toate cele 3 valori astfel:

```
int a, b, c;

buffer >> a >> b >> c;
```

Următorul program, THREEVAL.CPP, folosește operatorul de extracție pentru a citi 3 valori dintr-un buffer șir:

```
#include <iostream.h>
#include <strstream.h>

void main(void)
{
    char °ir = "1 100 1001";

    istream buffer(°ir);

    int a, b, c;

    buffer >> a >> b >> c;
```

```
cout << "The values are " << a << " " << b << " " << c << endl;
}
```

La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> THREEVAL <ENTER>
The values are 1 100 1001
```

În cazul precedent, programul a citit 3 variabile folosind o singură instrucțiune, dar și varianta următoare este corectă:

```
buffer >> a;
buffer >> b;
buffer >> c;
```

În anumite situații, nu se cunoaște numărul de valori conținute în buffer. Dacă programul încearcă să citească un buffer gol, stream-ul va activa indicatorul *fail*. Următorul program, STR\_FAIL.CPP, citește valori întregi dintr-un stream șir de intrare, până când stream-ul devine vid:

```
#include <iostream.h>
#include <strstream.h>

void main(void)
{
    char *oir = "1 2 3 4 5 6 7 8 9";

    int value;

    istrstream buffer(oir);

    while (!buffer.fail())
    {
        buffer >> value;
        if (!buffer.fail())
            cout << value << endl;
    }
}
```

După cum se poate vedea, programul ciclează până când se activează indicatorul *fail* al stream-ului. În cadrul ciclului, să observăm că programul testează starea indicatorului după fiecare operație de intrare. Dacă intrarea reușește, programul tipărește valoarea citită. La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> STR_FAIL <ENTER>
1
2
3
4
5
```

```
6
7
8
9
```

În programul precedent, buffer-ul conținea mai multe valori de tip *int*. În unele cazuri însă, este posibil ca buffer-ul să conțină valori de tipuri diferite. De exemplu, următorul program, DIFFTYPE.CPP, citește valori de tip *int*, *float* și *long* dintr-un stream șir de intrare:

```
#include <iostream.h>
#include <strstream.h>

void main(void)
{
    char *numbers = "1001 1.2345 123456789L";

    int count;
    float rate;
    long distance;

    istrstream buffer(numbers);

    buffer >> count;
    buffer >> rate;
    buffer >> distance;

    cout << "The int count is " << count << endl;
    cout << "The float rate is " << rate << endl;
    cout << "The long distance is " << distance << endl;
}
```

La compilarea și execuția acestui program, pe ecran, va apărea:

```
C:\> DIFFTYPE <ENTER>
The int count is 1001
The float rate is 1.2345
The long distance is 123456789
```

Când programele folosesc operatorul de extracție pentru a citi date dintr-un stream șir de intrare, compilatorul determină tipul valorii de extras după tipul variabilei. De exemplu, în programul precedent, compilatorul știa să extragă mai întâi o valoare de tipul *int*, urmată de *float* și *long*. Pentru a realiza aceste operații, compilatorul apelează o funcție care citește și convertește valoarea în tipul de date corect. În anumite situații, însă, valorile din buffer pot să nu corespundă celor pe care le necesită programul. De exemplu, să presupunem că, în loc să conțină valorile:

```
"1001 1.2345 123456789L"
```

buffer-ul începe cu un șir de caractere:

"ABC 1001 1.2345 123456789L"

Când funcția apelată pentru conversia unei valori de un anumit tip întâlnește un caracter incorect, ea va opri imediat conversia, returnând valoarea creată până la punctul de eroare. Funcția nu va avansa pointerul dincolo de punctul de eroare. Dacă primul caracter din stream este invalid, funcția va atribui indicatorul *fail* al stream-ului. Următorul program, BAD\_DATA.CPP, ilustrează modul de folosire a funcției *fail* pentru a determina dacă o operație I/O a reușit:

```
#include <iostream.h>
#include <strstream.h>

void main(void)
{
    char *numbers = "ABC 1001 1.2345 123456789L";

    int count;
    float rate;
    long distance;

    istrstream buffer(numbers);

    buffer >> count;

    if (buffer.fail())
    {
        cerr << "Error reading the int value for count" << endl;
        buffer.clear();
    }

    buffer >> rate;

    if (buffer.fail())
    {
        cerr << "Error reading the float value for rate" << endl;
        buffer.clear();
    }

    buffer >> distance;

    if (buffer.fail())
    {
        cerr << "Error reading the long value for distance" << endl;
        buffer.clear();
    }

    cout << "The int count is " << count << endl;
    cout << "The float rate is " << rate << endl;
    cout << "The long distance is " << distance << endl;
}
```

După cum se observă, programul folosește funcția membru *fail* pentru a testa reușita fiecărei operații I/O. Dacă are loc o eroare, programul afișează mesajul de eroare și apoi dezactivează indicatorul. Dacă o funcție de conversie întâlnește un caracter incorect, funcția oprește conversia fără a avansa pointerul la stream. Astfel, toate cele 3 operații de intrare au eșuat la întâlnirea literei A. Dacă compilați și executați acest program, pe ecran va apărea:

```
C:\> BAD_DATA <ENTER>
Error reading the int value for count
Error reading the float value for rate
Error reading the long value for distance
The int count is 0
The float rate is 1.3705e-034
The long distance is 0
```

Se observă că programul afișează un mesaj de eroare după fiecare operație I/O, iar valorile atribuite variabilelor nu sunt corecte.

#### CHEIA SUCCESULUI



#### CITIREA MAI MULTOR VALORI DINTR-UN STREAM ȘIR DE INTRARE

Un stream șir de intrare este un buffer de caractere din care programele pot citi valori folosind operatorul de extracție. În funcție de conținutul stream-ului, programele pot citi valori multiple dintr-un stream într-o singură instrucțiune:

```
buffer >> day >> month >> year;
```

Când compilatorul C++ întâlnește o operație I/O pe stream, el apelează o funcție pentru a extrage o valoare de un anumit tip. Dacă funcția se execută cu succes, ea convertește valoarea reprezentată în ASCII într-o valoare având tipul variabilei asociate. Dacă în timpul conversiei apare o eroare, funcția activează indicatorul *fail* al stream-ului și oprește imediat conversia, returnând valoarea creată până în acel moment. Funcția nu va avansa pointerul la stream dincolo de caracterul incorect.

#### FOLOSIREA FUNCȚIILOR MEMBRU ALE STREAM-ULUI ȘIR DE INTRARE

Programele anterioare au folosit operatorul de extracție pentru a citi valori dintr-un stream șir de intrare. Utilizarea cea mai frecventă a unui stream șir de intrare este de a converti o valoare din reprezentarea ei ASCII. Totuși, în funcție de necesitățile programului și conținutul stream-ului, pot exista situații când se dorește citirea unui singur caracter din stream, sau a unui buffer întreg. Următorul program, IN\_GET.CPP, citește conținutul unui stream caracter cu caracter, folosind funcția membru *get*:

```
#include <iostream.h>
#include <strstream.h>

void main(void)
{
    char *oir = "Success with C++";

    istrstream buffer(oir);

    for (char letter = buffer.get(); ! buffer.eof();
        letter = buffer.get())
        cout.put(letter);
}
```

Așa cum se vede, programul folosește funcția membru *get* pentru a citi un caracter din stream și funcția *eof* pentru a determina dacă stream-ul este vid.

În funcție de informația conținută în buffer-ul stream-ului, apare câteodată necesitatea de a citi informațiile situate după o serie de caractere, înainte de a începe o operație de extracție. Folosind funcția *get* în acest mod, se poate citi conținutul stream-ului caracter cu caracter. Într-un mod similar, următorul program, READALL.CPP, citește în totalitate buffer-ul unui stream folosind funcția membru *getline*:

```
#include <iostream.h>
#include <strstream.h>

void main(void)
{
    char *oir = "Success with C++";

    istrstream buffer(oir);

    char book[256];

    buffer.getline(book, sizeof(book));

    cout << book << endl;
}
```

### FOLOSIREA STREAM-URILOR ȘIR DE IEȘIRE

Așa cum convertim reprezentări ASCII în valori de tip *int* sau *float* folosind stream-uri șir de intrare, la fel putem converti valorile dintr-un șir. În limbajul C, programatorii folosesc funcții ca *itoa* (*int* în ASCII) pentru a converti o valoare numerică în ASCII. Următorul program, ITOA.C, ilustrează o astfel de conversie:

```
#include <stdio.h>
#include <stdlib.h>
```

```
void main(void)
{
    char buffer[256];

    int value = 1001;

    itoa(value, buffer, 10);

    printf("Value in ASCII is %s\n", buffer);
}
```

În alte programe C se folosește funcția *sprintf* pentru a crea o reprezentare ASCII a unei valori. Următorul program, SPRINTF.C, folosește funcția *sprintf* pentru a crea reprezentarea ASCII a valorii 1.2345:

```
#include <stdio.h>

void main(void)
{
    char buffer[256];

    float value = 1.2345;

    sprintf(buffer, "%f", value);

    printf("The value in ASCII is %s\n", buffer);
}
```

În limbajul C++, valorile pot fi convertite în ASCII folosind stream-urile șir de ieșire. Crearea unui stream șir de ieșire se face la fel cu cea a unui stream șir de intrare, declarând un obiect de tipul *ostrstream* la care se asociază un buffer, cum se arată mai jos:

```
char oir[256];

ostrstream buffer(oir, sizeof(oir));
```

Denumirea de *stream-uri șir de ieșire* derivă de la faptul că programele pot folosi operatorul de inserție pentru a plasa caractere ASCII în aceste stream-uri. Următorul program, OUT\_STR.CPP ilustrează modul în care s-ar putea folosi stream-urile șir de ieșire:

```
#include <strstream.h>
#include <iostream.h>
#include <iomanip.h>

void main(void)
{
    char oir[256];
    ostrstream buffer(oir, sizeof(oir));
```

```
float price = 29.95;
int programs = 300;

buffer << "Success with C++ $";
buffer << setprecision(2) << price << " with over ";
buffer << programs << " programs" << ends;

cout << "ir << endl;
}
```

Programul anterior folosește 3 instrucțiuni pentru a scrie caractere în stream-ul șir de ieșire. După cum se observă, instrucțiunile de ieșire folosesc manipulatorii *setprecision* și *ends*. Așa cum ați învățat, manipulatorul *setprecision* controlează numărul de cifre afișate la dreapta punctului zecimal. Manipulatorul *ends* inserează un caracter NULL în buffer-ul stream-ului, caracter ce marchează sfârșitul șirului. Stream-ul de ieșire din programul precedent a folosit un buffer de 256 octeți ce corespunde tabloului *șir*.

Când programele scriu date în stream-uri șir, bufferele acestor stream-uri nu pot primi un număr de caractere mai mare decât lungimea lor. Dacă un program încearcă să suprascrie un buffer, indicatorul *fail* al stream-ului este activat și caracterele următoare sunt ignorate. Următorul program, *OUT\_FAIL*, de exemplu, încearcă să scrie literele A până la Z într-un stream al cărui buffer are dimensiunea de 10 octeți. Folosind funcția membru *fail*, programul detectează eroarea și afișează un mesaj corespunzător:

```
#include <iostream.h>
#include <strstream.h>
#include <stdlib.h>

void main(void)
{
    char ir[10];

    ostrstream buffer(ir, sizeof(ir));

    for (char letter = 'A'; letter <= 'Z'; letter++)
    {
        buffer << letter;

        if (buffer.fail())
        {
            cerr << endl << "Error writing the character " <<
                letter << endl;
            exit(1);
        }
        else
            cout << letter;
    }
}
```

La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> OUT_FAIL <ENTER>
ABCDEFGHIJ
Error writing the letter K
```

### FOLOSIREA UNUI BUFFER DINAMIC

Programele anterioare au folosit un buffer de lungime fixă ce corespundea tabloului *șir*. Dacă la crearea unui stream șir de ieșire nu se specifică buffer-ul și dimensiunea sa, atunci funcția constructor *ostrstream* va alocă un buffer dinamic, folosind operatorul *new*. Următoarea instrucțiune, de exemplu, creează un stream șir de ieșire, denumit *buffer*, ce folosește un tablou dinamic:

```
ostrstream buffer;
```

Următorul program, *DYNAMIC.CPP*, creează un stream șir de ieșire a cărui dimensiune este dinamică. Programul scrie apoi literele alfabetului în buffer de 100 de ori (2600 de octeți). Folosind manipulatorul *ends*, programul oprește stream-ul și afișează literele prin transferul caracterelor către *cout*:

```
#include <iostream.h>
#include <strstream.h>

void main(void)
{
    ostrstream buffer;

    for (int count = 0; count < 100; count++)
        for (char letter = 'A'; letter <= 'Z'; letter++)
            buffer << letter;

    buffer << ends;

    cout << buffer.rdbuf();
}
```

După cum se observă, programul nu specifică buffer-ul de caractere care se folosește cu stream-ul. În schimb, *ostrstream* va alocă memoria dinamic, din zona liberă, folosind operatorul *new*. Pentru a afișa conținutul stream-ului, programul trebuie să apeleze buffer-ul ce conține caracterele. În acest scop, programul folosește funcția membru *rdbuf*. Funcția *rdbuf* returnează un pointer la începutul buffer-ului.

Când un stream de ieșire folosește un buffer dinamic, dimensiunea buffer-ului poate crește (în fundal) în timpul execuției programului. Stream-ul va alocă volumul de memorie necesar, eventual până la epuizarea spațiului liber de memorie. Următorul program, *EAT\_HEAP.CPP*, creează un stream șir de ieșire

ce folosește un buffer dinamic. Programul umple apoi buffer-ul cu literele alfabetului până la epuizarea spațiului de memorie liberă:

```
#include <iostream.h>
#include <strstream.h>

void main(void)
{
    ostrstream buffer;

    long count = 0L;

    cout << "Working...";

    while (!buffer.fail())
        for (char letter = 'A'; letter <= 'Z'; letter++, count++)
            buffer << letter;

    cout << "The number of characters buffered was " << count <<
        endl;
}
```



#### CREAREA UNUI STREAM ȘIR DE IEȘIRE DINAMIC

Pentru a crea un stream șir de ieșire, trebuie declarat un obiect de tipul *ostrstream*. În mod normal, programele atribuie un buffer de caractere stream-ului, după cum se vede mai jos:

```
char *ir[256];

ostrstream buffer(*ir, sizeof(*ir));
```

Dacă programul nu specifică un buffer, funcția constructor *ostrstream* va alocă automat zone din memoria disponibilă pentru buffer-ul stream-ului. Când programul atribuie buffer-ului dinamic un volum de caractere mai mare decât capacitatea acestuia, dimensiunea buffer-ului este mărită automat, prin alocarea de memorie din zona fragmentată. Pentru folosirea datelor din buffer-ul stream-ului, programul poate obține un pointer la începutul buffer-ului folosind funcția membru *rdbuf*, cum se arată mai jos:

```
cout << buffer.rdbuf();
```

#### FOLOSIREA FUNCȚIILOR MEMBRU ALE STREAM-ULUI ȘIR DE IEȘIRE

Câteva din programele precedente au folosit funcțiile *fail* și *rdbuf* cu stream-urile șir de ieșire. În funcție de cerințele programului, la un moment dat trebuie să se lucreze cu câte un caracter al buffer-ului, sau chiar cu întregul buffer.

Următorul program, OUT\_CHAR.CPP, folosește funcția membru *put* pentru a plasa în buffer literele alfabetului, una câte una:

```
#include <iostream.h>
#include <strstream.h>

void main(void)
{
    ostrstream buffer;

    for (char letter = 'A'; letter <= 'Z'; letter++)
        buffer.put(letter);

    buffer.put((char)NULL);

    cout << buffer.rdbuf() << endl;
}
```

Programele precedente au plasat caractere în stream-ul șir fără a lua în considerație numărul de caractere pe care buffer-ul îl poate conține. Dacă programul dorește să cunoască numărul de caractere din buffer, se poate folosi funcția membru *pcount*. Următorul program, PCOUNT.CPP, folosește această funcție pentru a afișa numărul de caractere conținute în buffer la diferite momente ale execuției programului:

```
#include <iostream.h>
#include <strstream.h>

void main(void)
{
    ostrstream buffer;

    cout << "The buffer currently contains " << buffer.pcount() <<
        " bytes " << endl;

    buffer << "Success with C++";

    cout << "The buffer currently contains " << buffer.pcount() <<
        " bytes " << endl;

    buffer << " " << 1 << " 2 " << 3 << endl << ends;

    cout << "The buffer currently contains " << buffer.pcount() <<
        " bytes " << endl;

    cout << buffer.rdbuf() << endl;
}
```

## BLOCAREA UNUI BUFFER DINAMIC PRIN FUNCȚIA STR

La folosirea unui stream șir de ieșire dinamic, funcția *rdbuf* este apelată pentru a obține un pointer la buffer-ul stream-ului. Se poate, însă, folosi și funcția membru *str* pentru a obține un pointer la buffer. Următorul program, *USE\_STR.CPP*, folosește această funcție pentru a afișa conținutul buffer-ului:

```
#include <iostream.h>
#include <strstream.h>

void main(void)
{
    ostrstream buffer;

    buffer << "Success with C++";

    cout << buffer.str() << endl;
}
```

La invocarea funcției *str*, programul blochează buffer-ul, iar acesta nu mai acceptă alte caractere. Și iată de ce: dimensiunea unui buffer dinamic crește în conformitate cu cerințele programului. Pentru a crește dimensiunea buffer-ului, funcțiile stream-ului deplasează părți ale buffer-ului în diferite locații de memorie. Funcțiile stream-ului pot deplasa buffer-ul, deoarece pentru program nu are importanță locația buffer-ului. Când este apelată funcția *str*, însă, funcțiile renunță la monopolul asupra buffer-ului, transferând programului gestiunea acestuia. Ca atare, funcțiile blochează buffer-ul, împiedicând creșterea sau deplasarea acestuia. O dată ce buffer-ul este blocat, inserarea altor caractere în buffer nu mai este permisă. În plus, când programul nu mai are nevoie de buffer, ar trebui folosit operatorul *delete* pentru eliberarea memoriei alocate pentru buffer. Din cauza acestor complexități, majoritatea programelor folosesc funcția *rdbuf* în locul funcției *str*.

## STREAM-URI ȘIR DE INTRARE/IEȘIRE

În Capitolul 3 ați învățat cum se deschid fișiere pentru operații de citire și scriere. Analog, pot fi realizate operații de citire și scriere pe stream-uri șir. Pentru a crea un stream șir capabil de a permite atât intrarea (extracția), cât și ieșirea (inserția), trebuie creat un obiect de tipul *strstream*, ca mai jos:

```
strstream buffer;
```

Următorul program, *INOUTSTR.CPP*, creează un stream șir pentru operații de intrare și ieșire. Programul trimite apoi în buffer o valoare reală, apoi introduce valoarea într-o variabilă de tip *float*:

```
#include <iostream.h>
#include <strstream.h>
```

```
void main(void)
{
    strstream buffer;

    float value;

    buffer << 1.2345;

    buffer >> value;

    cout << "The value is " << value << endl;
}
```

În funcție de cerințele unui program, poate fi necesară introducerea unor date utilizator, prelucrarea datelor și apoi atribuirea acestora unor variabile. De exemplu, programul ar putea mai întâi testa datele pentru a se asigura că acestea sunt numere sau text, înainte de a încerca asignarea acestor date unor variabile ale programului:

Când programele folosesc obiecte de tipul *strstream*, programul poate folosi buffere dinamice, cum s-a arătat mai înainte, sau poate specifica un buffer, după cum urmează:

```
char ir[64];

strstream buffer(ir, sizeof(ir));
```

În programe pot apărea situații când dorim să extindem un buffer, prin adăugarea altor caractere. De exemplu, buffer-ul ar putea conține inițial caracterele „Hello”, iar programul dorește să completeze acest text. Pentru a adăuga text la conținutul unui buffer se includ indicatorii *ios::ate* sau *ios::app*, cum se arată mai jos.

Următorul program, *HELLO.CPP*, adaugă text la un buffer ce conține cuvântul „Hello”:

```
#include <iostream.h>
#include <strstream.h>

void main(int argc, char **argv)
{
    char ir[256] = "Hello. ";

    strstream buffer(ir, sizeof(ir), ios::app);

    buffer << argv[1] << endl;

    cout << buffer.str();
}
```

Pentru a afișa un mesaj folosind programul *HELLO*, programul trebuie apelat împreună cu mesajul dorit, astfel:

```
C:\> HELLO world! <ENTER>
Hello, world!
```

## SĂ ÎNȚELEM RELAȚIILE DINTRE STREAM-URI

În Capitolul 11 ați examinat relația dintre diferite stream-uri I/O.

Clasele de stream șir definite în fișierul antet STRSTREA.H sunt, de asemenea, legate de celelalte clase de stream. Figura 12.3, de exemplu, ilustrează modul în care stream-urile șir sunt dependente de clasele I/O discutate în Capitolul 11:

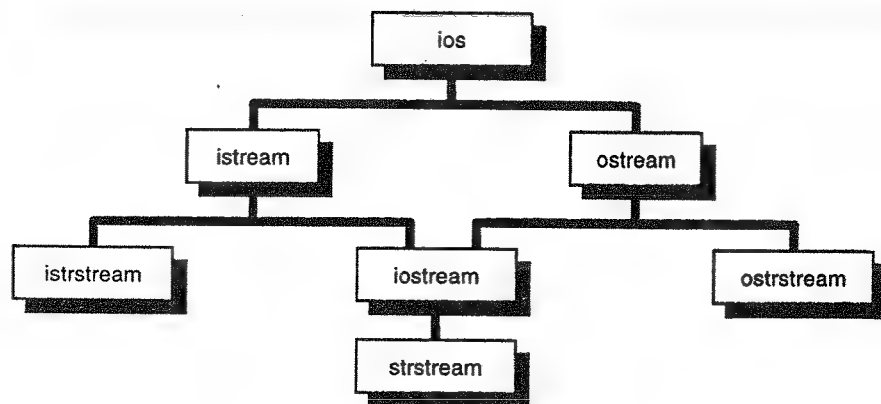


Figura 12.3 Relația dintre clasele stream I/O și clasele stream șir.

## REZUMAT

Un stream șir este o secvență de caractere conținută într-un buffer de memorie și folosită pentru conversii de date sau formatarea intrării/ieșirii. Acest capitol a examinat stream-urile șir și operațiile aferente în detaliu. Înainte de a trece la Capitolul 13, asigurați-vă că ați învățat următoarele:

- ✓ Un stream șir este un buffer de caractere plasat în memorie. Folosind operatorii de stream (operatorii de inserție și extracție), programele pot realiza operații de intrare/ieșire pe stream-uri șir.
- ✓ Fișierul antet STRSTREA.H conține definițiile de clase pentru stream-urile șir.
- ✓ Cele 3 clase importante pentru stream-urile șir sunt *istrstream*, *ostrstream* și *stringstream*. Clasa *istrstream* permite realizarea operațiilor de intrare pe stream. Clasa *ostrstream* permite efectuarea

operațiilor de ieșire, iar obiectele de tip *stringstream* permit operații de intrare și ieșire.

- ✓ Stream-urile șir de intrare sunt astfel denumite deoarece programele pot folosi operatorul de extracție (>>) sau funcțiile membru ale stream-ului pentru a introduce valori din stream și a le atribui variabilelor.
- ✓ Stream-urile șir de ieșire sunt astfel denumite deoarece programele pot folosi operatorul de inserție (<<) sau funcțiile membru ale stream-ului pentru a extrage valori din stream.
- ✓ Deoarece stream-urile șir sunt definite de clase, ele acceptă funcții membru. Dacă examinați fișierul antet STRSTREA.H, veți găsi funcții membru corespunzătoare funcțiilor I/O discutate în Capitolul 1.
- ✓ Când creați obiecte stream șir de ieșire, programele pot specifica un buffer în care stream-ul va memora datele. De asemenea, programele pot dirija stream-ul pentru a folosi un buffer dinamic, alocat de funcțiile stream-ului din zona de memorie liberă. Dacă se folosește buffer dinamic, dimensiunea acestuia poate crește (în fundal) în cursul execuției programului.



## CAPITOLUL 13

### APROFUNDAREA FUNCȚIILOR VIRTUALE

În Capitolul 8 ați învățat cum se folosesc funcțiile virtuale în C++ pentru a realiza polimorfismul. Tot în Capitolul 8 ați citit că funcțiile virtuale sunt posibile datorită operațiilor „discrete” efectuate de compilator. Acest capitol examinează câteva din aceste operații în încercarea de a clarifica, în parte, câteva din enigmele ce însoțesc funcțiile virtuale. Când veți termina acest capitol, veți cunoaște următoarele:

- ♦ Ce sunt legăturile statice și dinamice
- ♦ Ce este o tabelă de funcții virtuale
- ♦ Prețul plătit pentru flexibilitatea sporită generată de funcțiile virtuale

#### ANALIZA UNUI EXEMPLU SIMPLU

Așa cum ați învățat în Capitolul 8, polimorfismul permite unui obiect să ia două sau mai multe forme. Următorul program, SHAPES.CPP, folosește funcții virtuale pentru a crea un pointer la obiectul polimorfic *shape*. Începem cu crearea următoarei clase de bază *shape*:

```
class shape {
public:
    shape(char *name, int x, int y);
    virtual void draw(void);
protected:
    char name[64];
    int x;
    int y;
};
```

Se observă că programul declară funcția *draw* ca fiind virtuală. Programul derivează apoi obiectele *circle* și *square*, astfel:

```
class circle : public shape {
public:
    circle(char *name, int x, int y, int radius);
    void draw(void);
private:
    char name[64];
    int radius;
};
```

```
class square : public shape {
public:
    square(char *name, int x, int y, int side);
private:
    char name[64];
    int side;
};
```

Folosind funcțiile virtuale, programul va atribui pointerului obiect *graphic* diferite forme (figuri geometrice). Următoarele instrucțiuni implementează programul SHAPES.CPP:

```
#include <iostream.h>
#include <string.h>
```

```
class shape {
public:
    shape(char *name, int x, int y);
    virtual void draw(void);
protected:
    char name[64];
    int x;
    int y;
};
```

```
class circle : public shape {
public:
    circle(char *name, int x, int y, int radius);
    void draw(void);
private:
    char name[64];
    int radius;
};
```

```
class square : public shape {
public:
    square(char *name, int x, int y, int side);
private:
    char name[64];
    int side;
};
```

```
shape::shape(char *name, int x, int y)
{
    strcpy(shape::name, name);
    shape::x = x;
    shape::y = y;
}
```

```
void shape::draw(void)
{
```

```
    cout << "Drawing the shape: " << name << endl;
}
```

```
circle::circle(char *name, int x, int y, int radius) :
    shape(name, x, y)
{
    circle::radius = radius;
}
```

```
void circle::draw(void)
{
```

```
    cout << "Circles are being drawn: ";
    cout << "x " << x << " y " << y << " radius " << radius << endl;
}
```

```
square::square(char *name, int x, int y, int side) :
    shape(name, x, y)
{
    square::side = side;
}
```

```
void main(void)
{
```

```
    circle ball("Circle", 10, 20, 30);
```

```
    square box("Square", 40, 50, 40);
```

```
    shape *graphic = &ball;
```

```
    graphic->draw();
```

```
    graphic = &box;
```

```
    graphic->draw();
}
```

Așa cum se observă, pointerul *graphic* ia două forme, *circle* și *square*.

La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> SHAPES <ENTER>
Circles are being drawn: x 10 y 20 radius 30
Drawing the shape: Square
```

### POLIMORFISMUL NU ÎNSEAMNĂ SUPRAPUNEREA FUNCȚIILOR

Așa cum ați învățat în Capitolul 8, polimorfismul nu este același lucru cu suprapunerea funcțiilor. De exemplu, următorul program, SUMARRAY.CPP, suprapune funcția *sum\_array*:

```
#include <iostream.h>

float sum_array(float *array, int num_elements)
{
    float sum = 0.0;

    for (int i = 0; i < num_elements; i++)
        sum += array[i];

    return(sum);
}

long sum_array(int *array, int num_elements)
{
    long sum = 0L;

    for (int i = 0; i < num_elements; i++)
        sum += array[i];

    return(sum);
}

void main(void)
{
    int int_array[5] = {1, 2, 3, 4, 5};
    float float_array[5] = {1.1, 2.2, 3.3, 4.4, 5.5};

    cout << "Values in int array: " << sum_array(int_array, 5) << endl;
    cout << "Values in float array: " << sum_array(float_array, 5) << endl;
}
```

La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> SUMARRAY <ENTER>
Values in int array: 15
Values in float array: 16.5
```

În timpul compilării, compilatorul C++ determină care din funcții va fi folosită, bazându-se pe parametri efectiv transferați funcțiilor sau pe tipul valorilor returnate de aceste funcții. Compilatorul va substitui atunci funcția corectă pentru fiecare apelare. Astfel de substituții efectuate la compilare se numesc *legături statice*. Ele sunt efectuate *înainte* de execuția programului. Când într-un program se realizează polimorfism, pe de altă parte, substituțiile sunt efectuate *în timpul* execuției programului. Astfel de substituții la rulare se numesc *legături dinamice*.

### SĂ ÎNȚELEM LEGĂTURILE DINAMICE

Din punctul de vedere al compilatorului, suprapunerea funcțiilor este o sarcină ușoară, deoarece acesta cunoaște modul de utilizare a fiecărui program înainte de lansarea în execuție. În cazul următorului program, însă, obiectul care este atribuit pointerului obiect *graphic* nu este cunoscut anterior execuției programului. Programul ASKSHAPE.CPP cere utilizatorului să selecteze obiectul *circle* sau *square*. În funcție de selecția utilizatorului, programul va atribui pointerului unul din obiecte:

```
#include <ctype.h>

void main(void)
{
    circle ball("Circle", 10, 20, 30);

    square box("Square", 40, 50, 40);

    shape *graphic;

    char choice;

    do {
        cout << "Type C for circle S for square: ";
        cin >> choice;
        choice = toupper(choice);

        if (choice == 'S')
            graphic = &box;
        else if (choice == 'C')
            graphic = &ball;
    } while ((choice != 'C') && (choice != 'S'));

    cout << endl << endl;

    graphic->draw();
}
```

**Observație:** Programul ASKSHAPE.CPP nu a inclus și declarațiile de clasă și funcțiile membru.

Așa cum se poate observa, în funcție de opțiunea utilizatorului, programul atribuie diferiți pointeri obiect la pointerul *graphic*. Cu alte cuvinte, programul trebuie să determine adresa corectă a obiectului în timpul execuției. Pentru a determina astfel de adrese, programul folosește intrinsec o *tabelă de funcții virtuale*. Această tabelă este construită de compilator.

## SĂ ÎNȚELEM TABELELE DE FUNCȚII VIRTUALE

O tabelă de funcții virtuale este o tabelă pe care C++ o folosește la execuția programului pentru a localiza funcția membru corectă, în cazul utilizării polimorfismului. Tabela conține rubrici pentru obiectele ce folosesc clase virtuale. Când programul folosește o astfel de funcție, C++ caută în tabelă adresa funcției corecte, apoi apelează funcția.

De fiecare dată când o clasă folosește una sau mai multe funcții virtuale, compilatorul C++ îi adaugă un pointer special ce indică tabela virtuală. Următorul program, TABLEPTR.CPP, folosește operatorul *sizeof* pentru a afișa dimensiunea a două clase foarte asemănătoare. Singura diferență între cele două clase este aceea că prima dintre ele folosește o funcție virtuală. Ca urmare, prima clasă conține un pointer special la tabela de funcții virtuale:

```
#include <iostream.h>

class shape1 {
public:
    shape(char *name, int x, int y);
    virtual void draw(void);
protected:
    char name[64];
    int x;
    int y;
};

class shape2 {
public:
    shape(char *name, int x, int y);
    void draw(void);
protected:
    char name[64];
    int x;
    int y;
};

void main(void)
{
    cout << "Size of virtual class: " << sizeof(shape2) << endl;
    cout << "Size of nonvirtual class: " << sizeof(shape1) << endl;
}
```

La compilarea și execuția acestui program în Borland C++, pe ecran va apărea:

```
C:\> TABLEPTR <ENTER>
Size of virtual class: 68
Size of nonvirtual class: 70
```

După cum se observă, clasa care folosește funcția virtuală este mai cuprinzătoare, datorită pointerului la tabela virtuală.

## SUPRASOLICITAREA GENERATĂ DE FUNCȚIILE VIRTUALE

Deși funcțiile virtuale aduc o mare flexibilitate în programe, folosirea lor nu se face fără un anumit efort de timp și de memorie. De fiecare dată când programul folosește un obiect ce conține una sau mai multe funcții virtuale, este folosit un pointer special pentru accesul la tabela virtuală, care va indica, la rândul ei, adresa funcției corecte. Deoarece apelarea unei funcții virtuale cere o indirectare suplimentară a memoriei, ea se va derula mai încet decât apelarea unei funcții standard. Următorul program, OVERHEAD.CPP, apelează două funcții similare de 5 milioane de ori, măsurând timpul necesar apelării fiecărei funcții. Prima funcție este virtuală, în timp ce a doua este o funcție standard:

```
#include <iostream.h>
#include <string.h>
#include <time.h>

class shape {
public:
    shape(int x) { shape::x = x; };
    virtual void draw(void);
protected:
    int x;
};

class circle {
public:
    circle(int x) { circle::x = x; };
    void draw(void);
private:
    int x;
};

void shape::draw(void)
{
    return ;
}

void circle::draw(void)
{
    return ;
}

void main(void)
{
    time_t start_time, stop_time;

    circle ball(1);

    shape box(2);
```

```

long int i;

cout << "Working..." << endl;

time(&start_time);
for (i = 0; i < 5000000L; i++)
    box.draw();
time(&stop_time);

cout << "Virtual required: " << stop_time - start_time << endl;

time(&start_time);
for (i = 0; i < 5000000L; i++)
    ball.draw();
time(&stop_time);

cout << "Standard required: " << stop_time - start_time << endl;
}

```

În funcție de calculatorul folosit, timpul necesar fiecărei funcții poate diferi. Totuși, dată fiind viteza foarte mare a procesoarelor de astăzi, consumul suplimentar asociat cu funcțiile virtuale devine minimal pentru majoritatea aplicațiilor și poate fi neglijat.

### SĂ ÎNȚELEM DESTRUCȚORII VIRTUALE

În Capitolul 2 s-au prezentat funcțiile constructor și destructor. Toate programele din această carte au făcut uz de funcțiile constructor, iar unele au folosit și funcțiile destructor. Când definiți funcții constructor și destructor, rețineți următoarele: *funcțiile constructor nu pot fi virtuale, dar funcțiile destructor pot fi virtuale.*

Așa cum ați învățat, un obiect polimorfic poate lua una sau mai multe forme. Când obiectul corespunzător este distrus, programul trebuie să apeleze destructorul corect. De exemplu, să considerăm următorul program, BADDESTR.CPP, care folosește pointerul polimorfic *graphic*. Programul definește o funcție destructor pentru fiecare tip de obiect, care afișează un mesaj ce informează utilizatorul despre apelarea destructorului.

```

#include <iostream.h>
#include <string.h>

class shape {
public:
    shape(char *name, int x, int y);
    virtual void draw(void);
    ~shape(void);
protected:
    char name[64];

```

```

    int x;
    int y;
};

class circle : public shape {
public:
    circle(char *name, int x, int y, int radius);
    void draw(void);
    ~circle(void);
private:
    char name[64];
    int radius;
};

shape::shape(char *name, int x, int y)
{
    strcpy(shape::name, name);
    shape::x = x;
    shape::y = y;
}

void shape::draw(void)
{
    cout << "Drawing the shape: " << name << endl;
}

shape::~shape()
{
    cout << "In shape's destructor" << endl;
}

circle::circle(char *name, int x, int y, int radius) :
    shape(name, x, y)
{
    circle::radius = radius;
}

void circle::draw(void)
{
    cout << "Circles are being drawn: ";
    cout << "x " << x << " y " << y << " radius " << radius << endl;
}

circle::~circle(void)
{
    cout << "In circle's destructor" << endl;
}

void main(void)

```

```
{
    shape *graphic = new circle("Circle", 10, 20, 30);

    graphic->draw();
    delete graphic;
}
```

Așa cum se observă, programul creează un obiect *circle*, atribuind un pointer al obiectului către variabila polimorfică *graphic*. După ce programul desenează obiectul, se folosește operatorul *delete* pentru distrugerea obiectului. Din păcate, deoarece *graphic* este un pointer la clasa *shape*, programul invocă destructorul clasei *shape*, și nu destructorul clasei *circle*. Drept rezultat, când compilați și executați acest program, pe ecran va apărea:

```
C:\> BADDESTR <ENTER>
Circles are being drawn: x 10 y 20 radius 30
In shape's destructor
```

În continuare, să presupunem că clasa *circle* a alocat memorie care trebuie disponibilizată de către o funcție destructor. După cum ați văzut, programul nu apelează destructorul clasei *circle*.

Totuși, dacă se transformă funcția destructor a clasei *shape* într-o funcție virtuală, programul va apela, consecutiv, constructorii claselor *circle* și *shape*, cum se arată mai jos:

```
C:\> BADDESTR <ENTER>
Circles are being drawn: x 10 y 20 radius 30
In circle's destructor
In shape's destructor
```

După cum se observă, programul apelează corect fiecare din funcțiile destructor. Când un program folosește polimorfismul și diferite clase alocă memorie, se recomandă folosirea funcțiilor destructor virtuale pentru eliberarea corectă a memoriei.

## REZUMAT

În Capitolul 8 ați examinat polimorfismul și funcțiile virtuale în detaliu. În acest capitol ați învățat câteva din operațiile intrinseci pe care le efectuează C++ pentru a implementa funcțiile virtuale. Înainte de a trece la Capitolul 15, asigurați-vă că ați învățat următoarele:

- ✓ Legăturile statice sunt substituții efectuate de compilatorul C++ înainte de execuția programului, pentru a determina ce funcții trebuie apelate.
- ✓ Legăturile dinamice sunt substituții făcute în momentul execuției programului, când acesta realizează polimorfismul.

- ✓ O tabelă de funcții virtuale conține rubrici pentru obiecte ce folosesc clase virtuale.
- ✓ Deoarece apelarea unei funcții virtuale necesită o indirectare suplimentară a memoriei pentru a avea acces la tabela de funcții virtuale, aceasta se derulează mai încet decât apelarea unei funcții standard.
- ✓ Când într-un program se realizează polimorfismul și clasele acestuia alocă memoria dinamic, trebuie folosite funcții virtuale pentru eliberarea corectă a memoriei.

## CE SUNT CAZURILE DE EXCEPȚIE

O excepție este o eroare definită de program. Dacă o funcție lucrează, de exemplu, cu tablouri, o excepție ar putea apărea dacă programul transferă funcției un index ce depășește limitele tabloului. Când testează valoarea indexului, funcția poate genera sau *lansa* o excepție. Analog, dacă o funcție adună două valori întregi, se poate genera o excepție dacă rezultatul depășește domeniul de valori întregi. Singurele excepții care apar în cadrul unui program sunt cele definite de program sau de bibliotecile folosite de program. Evenimentele hard, precum defectul de pagină, eroare la citirea pe disc sau alte întreruperi, nu generează excepții.

Pentru a realiza tratarea erorilor, se folosesc în programe cuvintele cheie *try*, *throw* și *catch*. Să presupunem, de exemplu, că programul folosește funcția *fill\_array* pentru a citi valori dintr-un fișier și a le insera într-un tablou de dimensiune fixă. Dacă fișierul conține mai multe valori decât poate memora tabloul, funcția ar trebui să genereze o excepție.

Înainte să apeleze funcția, programul folosește instrucțiunea *try* pentru a permite tratarea excepțiilor:

```
try {
    fill_array(some_array, 100);
}
```

După ce programul activează tratarea excepțiilor, se poate încerca detecția anumitor excepții folosind instrucțiunea *catch*, cum se indică mai jos:

```
try {
    fill_array(some_array, 100);
}

catch (array_overflow) {
    cerr << "Too many values for array to handle" << endl;
    exit(1);
}

catch (file_not_found) {
    cerr << "Could not find the data file" << endl;
    exit(1);
}
```

În exemplul precedent, programul apelează funcția *fill\_array* în cadrul instrucțiunii *try*. Apoi, folosind două instrucțiuni *catch*, programul va încerca să detecteze posibila apariție a două erori. Dacă apare oricare din cele două excepții, programul va efectua instrucțiunile corespunzătoare (rutina specifică de tratare a excepției). Numai o singură excepție poate apărea la un moment dat. Cu alte cuvinte, programul anterior nu poate genera în același timp excepțiile *array\_overflow* și *file\_not\_found*.

## PRELUAREA EXCEPȚIILOR ÎNTR-UN PROGRAM



Pentru a detecta și manipula o excepție într-un program, trebuie folosite instrucțiunile *try* și *catch*. Pentru a activa tratarea excepțiilor în cazul unei anumite secvențe de operații, se folosește instrucțiunea *try*:

```
try {
    some_operation(a, b, c);
}
```

Pentru a determina apoi dacă o excepție a avut loc, se folosește instrucțiunea *catch*:

```
catch (exception_1) {
    // instructiuni
}

catch (exception_2) {
    // instructiuni
}
```

## CUM SE DENUMESC EXCEPȚIILE

Fiecare excepție folosită într-un program are nume unic. Excepțiile se denumesc prin crearea claselor de excepții. De exemplu, următoarele instrucțiuni definesc 3 clase de excepții:

```
class open_error {};
class read_error {};
class write_error {};
```

În acest caz, clasele de excepții nu au membri. În exemplele următoare, veți vedea cum variabilele membru permit claselor de excepții să returneze mai multe informații despre excepție. Pentru a genera o excepție, programul trebuie să o lanseze cu ajutorul instrucțiunii *throw*. De exemplu, următoarea instrucțiune lansează excepția *write\_error*:

```
throw write_error(); // Include parenthesis after the class name
```

În acest mod, excepțiile sunt lansate și ulterior preluate de rutina de tratare folosind instrucțiunea *catch*. Următorul program, *COPY\_EXP.CPP*, folosește excepțiile pentru a determina apărute la deschiderea, citirea și scrierea fișierelor. Dacă o astfel de eroare apare în funcția *perform\_copy*, funcția va lansa excepția corespunzătoare. Folosind instrucțiunea *catch*, programul preia fiecare tip de excepție:

```
#include <fstream.h>
#include <stdlib.h>
```

```

class open_error { };
class read_error { };
class write_error { };

void perform_copy(char *source_file, char *target_file)
{
    ifstream source(source_file, ios::binary);

    char line[1];

    if (source.fail())
        throw open_error();
    else
    {
        ofstream target(target_file, ios::binary);

        if (target.fail())
            throw open_error();
        else
        {
            while (! source.eof() && ! source.fail())
            {
                source.read(line, sizeof(line));

                if (source.good())
                {
                    target.write(line, sizeof(line));

                    if (target.fail())
                        throw write_error();
                }
                else if (! source.eof())
                    throw read_error();
            }
            source.close();
            target.close();
        }
    }
}

void main(int argc, char **argv)
{
    try {
        perform_copy(argv[1], argv[2]);
    }
    catch (open_error) {
        cerr << "Error opening file" << endl;
        exit(1);
    }
}

```

Definirea claselor de excepții

Generarea unei excepții

Preluarea unei excepții

```

catch (read_error) {
    cerr << "Error reading file" << endl;
    exit(1);
}

catch (write_error) {
    cerr << "Error writing file" << endl;
    exit(1);
}

```

Preluarea unei excepții

**CHEIA SUCCESULUI**



**DENUMIREA ȘI GENERAREA EXCEPȚIILOR**

Fiecare excepție dintr-un program are nume unic. O excepție se definește prin crearea unei clase de excepții de forma:

```
class out_of_memory { };
```

Pentru a activa ulterior excepția, programul folosește cuvântul cheie *throw* și numele excepției, cum se arată mai jos:

```
throw out_of_memory();
```

Observați parantezele ce urmează după numele excepției în instrucțiunea *throw*. În acest caz, excepția va fi tratată de codul ce preia excepția *out\_of\_memory*:

```

try {
    some_operation()
}
catch (out_of_memory) {
    // Statements
}

```

**EXCEPȚII LOCALE ALE UNEI CLASE**

În exemplul anterior, clasele de excepții au fost globale pentru întregul program. În multe alte situații, însă, excepțiile sunt locale unei clase. De exemplu, să presupunem că un program folosește o clasă tablou pentru a memora valori întregi:

```

class Array {
public:
    Array(int size);
    void fill_array(int num_values);
    void show_array(int num_values);

private:
    int *buffer;
}

```



```
int size;
};
```

Când în program se creează un obiect de tip *Array*, trebuie specificată o dimensiune a bufferului, astfel:

```
Array vector(100);
```

În acest caz, programul va alocă un buffer capabil de a memora 100 de valori întregi. Funcția *fill\_array* va permite programului să atribuie valori consecutive elementelor tabloului. De exemplu, următoarea instrucțiune atribuie valorile 1 până la 25 primelor 25 de elemente (restul elementelor rămâne neschimbat):

```
vector.fill_array(25);
```

În mod similar, funcția *show\_array* permite afișarea unui anumit număr de elemente ale tabloului. Următoarea instrucțiune, de exemplu, afișează valorile primelor 10 elemente:

```
vector.show_array(10);
```

Următorul program, *ARRAY.CPP*, folosește clasa *Array* pentru a crea un vector de valori întregi:

```
#include <iostream.h>

class Array {
public:
    Array(int size);
    void fill_array(int num_values);
    void show_array(int num_values);

private:
    int *buffer;
    int size;
};

Array::Array(int size)
{
    buffer = new int[size];
    Array::size = size;
}

void Array::fill_array(int num_values)
{
    for (int i = 0; i < num_values; i++)
        buffer[i] = i;
}

void Array::show_array(int num_values)
{
}
```

```
for (int i = 0; i < num_values; i++)
    cout << buffer[i] << endl;
}
```

```
void main(void)
{
    Array vector(200);

    vector.fill_array(10);

    vector.show_array(10);
}
```

După cum se observă, programul folosește tabloul fără a testa valorile din indexul transmis funcțiilor *fill\_array* și *show\_array*. Să presupunem, de exemplu, că programul apelează funcția *fill\_array* cu valoarea de index 500:

```
vector.fill_array(500);
```

În acest caz, programul va atribui tabloului valorile de la 1 la 500, depășind tabloul cu 300 de valori. O variantă mai bună a programului ar trebui să testeze valoarea indexului, pentru a se asigura că aceasta este corectă, înainte de atribuirea valorilor întregi.

Următorul program, *ARRAYEXC.CPP*, folosește tratarea excepțiilor pentru a semnaliza o operație ce poate depăși limitele tabloului. Deoarece excepțiile sunt specifice clasei *Array*, domeniul excepțiilor este limitat la această clasă, definind excepțiile ca membri ai clasei:

```
#include <iostream.h>
#include <stdlib.h>

class Array {
public:
    Array(int size);
    void fill_array(int num_values);
    void show_array(int num_values);
    class range { };
private:
    int *buffer;
    int size;
};

Array::Array(int size)
{
    buffer = new int[size];
    Array::size = size;
}

void Array::fill_array(int num_values)
{
}
```

```

    if (num_values > size)
        throw range();

    for (int i = 0; i < num_values; i++)
        buffer[i] = i;
}

void Array::show_array(int num_values)
{
    if (num_values > size)
        throw range();

    for (int i = 0; i < num_values; i++)
        cout << buffer[i] << endl;
}

void main(void)
{
    Array vector(200);

    try {
        vector.fill_array(10);
    }
    catch (Array::range) {
        cerr << "Invalid array index in fill_array" << endl;
        exit(1);
    }

    try {
        vector.show_array(500);
    }
    catch (Array::range) {
        cerr << "Invalid array index in show_array" << endl;
        exit(1);
    }
}

```

După cum se observă, programul apelează funcția *show\_array* cu o valoare incorectă a indexului, și anume 500. Când funcția va examina această valoare, ea va lansa excepția, care este preluată de instrucțiunea *catch*.

## TRATAREA EXCEPȚIILOR

Când apare o excepție, programul trebuie să determine modul în care aceasta va fi tratată. În cazul programelor precedente se afișau numai mesaje de erori, urmate de încheierea programului. Alte programe pot încerca să rezolve erorile. Să presupunem, de exemplu, că un program primește o excepție de tip memorie insuficientă. În loc de a-și încheia execuția, programul ar putea elibera memoria ocupată sau ar putea colecta resturile de memorie fragmentată, cum se

va discuta în Capitolul 15. În astfel de cazuri, rutina de tratare va avea nevoie de acces la pointerii a căror memorie poate fi disponibilizată sau funcția trebuie să fie suficient de complicată pentru a traversa zona de memorie fragmentată, disponibilizând locațiile neesențiale. După ce memoria cerută de program devine disponibilă, programul poate repeta operația care a cauzat eroarea.

În alte situații, se poate dori pur și simplu terminarea programului. Dacă un program generează o excepție și aceasta nu este preluată, programul va lansa o rutină de tratare prestabilită care va încheia în mod automat programul. Următorul program, *BAD\_EXC.CPP*, lansează excepția *no\_such\_device*. Deoarece nu există cod care să preia excepția, programul se va termina pur și simplu:

```

#include <iostream.h>

class no_such_device { }; // Exception class
class file_not_found { };

void show_files(char disk_drive)
{
    if ((disk_drive < 'A') || (disk_drive > 'Z'))
        throw no_such_device();
    else
        cout << "Would display files here" << endl;
}

void main(void)
{
    try {
        show_files('a');
    }
    catch (file_not_found) {
        cerr << "Error finding file" << endl;
    }
}

```

În cazul precedent, programul detectează excepția *file\_not\_found*, nu și excepția *no\_such\_device*. Deoarece excepția lansată nu este preluată, programul se încheie.

## INFORMAȚII SUPLIMENTARE DESPRE O EXCEPȚIE

Când se generează excepții, uneori este necesară includerea unor informații suplimentare despre cauza erorilor. De exemplu, programul *COPY\_EXP.CPP*, prezentat anterior, lansează excepția *open\_error* când nu poate deschide fișierul sursă sau fișierul destinație. Din păcate, codul de program care preia excepția nu poate determina care din cele două fișiere nu a putut fi deschis. O soluție mai bună ar fi să se returneze o valoare ce poate fi testată de program, cum ar fi valoarea 0 pentru fișierul sursă și valoarea 1 pentru fișierul destinație.

O astfel de facilitare poate fi realizată ușor prin adăugarea unei variabile membru la clasa de excepții care să memoreze valoarea respectivă. În al doilea rând, trebuie creată o funcție membru a clasei ce poate fi lansată cu valoarea corespunzătoare. Următoarele instrucțiuni ilustrează modul în care se poate modifica clasa de excepții *open\_file*:

```
class open_error {
public:
    int value;
    open_error(int value) { open_error::value = value; };
};
```

După cum se poate observa, clasa conține variabila membru *value* și o funcție constructor ce atribuie acesteia valoarea specificată. La lansarea excepției, programul poate specifica o valoare drept parametru, cum se arată mai jos:

```
throw open_error(1); // Error opening target file
```

În acest caz, instrucțiunea *throw* determină crearea unui obiect de tip excepție, iar la execuția funcției constructor a obiectului, valoarea parametrului va fi atribuită variabilei membru. Când programul preia excepția, el primește de fapt obiectul eroare și prin urmare poate examina variabilele membru, cum se indică în continuare:

```
catch (open_error exception) {
    if (exception.value)
        cerr << "Error opening target file" << endl;
    else
        cerr << "Error opening source file" << endl;
    exit(1);
}
```

Pentru a se referi la obiectul *open\_error*, programul folosește o variabilă denumită *exception*. Testând valoarea variabilei membru *value*, programul poate determina dacă eroarea este cauzată de fișierul sursă sau fișierul destinație. Următorul program, EXPVALUE.CPP, folosește o variabilă membru pentru a determina care fișier cauzează eroarea:

```
#include <fstream.h>
#include <stdlib.h>

class open_error {
public:
    int value;
    open_error(int value) { open_error::value = value; };
};

class read_error {};
class write_error {};
```

```
void perform_copy(char *source_file, char *target_file)
{
    ifstream source(source_file, ios::binary);

    char line[1];

    if (source.fail())
        throw open_error(0);
    else
    {
        ofstream target(target_file, ios::binary);

        if (target.fail())
            throw open_error(1);
        else
        {
            while (! source.eof() && ! source.fail())
            {
                source.read(line, sizeof(line));

                if (source.good())
                {
                    target.write(line, sizeof(line));

                    if (target.fail())
                        throw write_error();
                }
                else if (! source.eof())
                    throw read_error();
            }
            source.close();
            target.close();
        }
    }
}
```

```
void main(int argc, char **argv)
{
    try {
        perform_copy(argv[1], argv[2]);
    }

    catch (open_error exception) {
        if (exception.value)
            cerr << "Error opening target file" << endl;
        else
            cerr << "Error opening source file" << endl;
        exit(1);
    }
}
```

```

catch (read_error) {
    cerr << "Error reading file" << endl;
    exit(1);
}

catch (write_error) {
    cerr << "Error writing file" << endl;
    exit(1);
}
}

```

În mod analog, următorul program, EXSTRING.CPP, modifică programul precedent prin includerea în clasa de excepții a numelui fișierului sub formă de șir de caractere:

```

#include <fstream.h>
#include <stdlib.h>
#include <string.h>

class open_error {
public:
    int value;
    char filename[64];
    open_error(int value) {
        open_error::value = value;
        filename[0] = NULL;
    };
    open_error(int value, char *filename) {
        open_error::value = value;
        strcpy(open_error::filename, filename);
    };
};

class read_error { };
class write_error { };

void perform_copy(char *source_file, char *target_file)
{
    ifstream source(source_file, ios::binary);

    char line[1];

    if (source.fail())
        throw open_error(0, source_file);
    else
    {
        ofstream target(target_file, ios::binary);

        if (target.fail())
            throw open_error(1, target_file);
        else

```

```

{
    while (! source.eof() && ! source.fail())
    {
        source.read(line, sizeof(line));

        if (source.good())
        {
            target.write(line, sizeof(line));

            if (target.fail())
                throw write_error();
        }
        else if (! source.eof())
            throw read_error();
    }
    source.close();
    target.close();
}
}

void main(int argc, char **argv)
{
    try {
        perform_copy(argv[1], argv[2]);
    }

    catch (open_error exception) {
        if (exception.value)
        {
            if (exception.filename)
                cerr << "Error opening the target file " <<
                    exception.filename << endl;
            else
                cerr << "Error opening target file" << endl;
        }
        else
        {
            if (exception.filename)
                cerr << "Error opening the source file " <<
                    exception.filename << endl;
            else
                cerr << "Error opening source file" << endl;
        }
    }

    exit(1);
}

catch (read_error) {
    cerr << "Error reading file" << endl;
    exit(1);
}

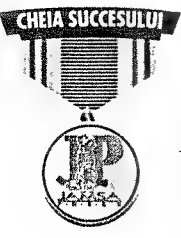
```

```

    }

    catch (write_error) {
        cerr << "Error writing file" << endl;
        exit(1);
    }
}

```



**RETURNAREA UNOR VALORI ALĂTURI DE EXCEPȚIE**

La generarea excepțiilor în programe, se pot specifica informații suplimentare despre excepții. Acest lucru se poate realiza prin includerea unor variabile membru în clasa de excepții și a unor funcții constructor ce pot atribui acestor variabile valorile unor parametri. Când programul preia excepția, el poate examina variabilele membru ale excepției pentru a găsi mai multe informații despre cauza acesteia.

### PRELUAREA MAI MULTOR EXCEPȚII ÎNTR-O SINGURĂ INSTRUCȚIUNE CATCH

Așa cum ați învățat, dacă programul nu preia o anumită excepție, se va apela o rutină prestabilită și programul se va încheia. Drept urmare, programele pot avea o serie de instrucțiuni *catch*. În funcție de modul de tratare al excepțiilor, rutinele de tratare ar putea fi similare celor prezentate mai jos:

```

catch (file_open_error) {
    cerr << "Error opening file" << endl;
    exit(1);
}
catch (source_file_open_error) {
    cerr << "Error opening file" << endl;
    exit(1);
}
catch (target_file_open_error) {
    cerr << "Error opening file" << endl;
    exit(1);
}
catch (temp_open_error) {
    cerr << "Error opening file" << endl;
    exit(1);
}

```

După cum se poate observa, cele 4 rutine de tratare a excepțiilor realizează operații similare. În aceste situații, când excepțiile sunt oarecum asemănătoare, s-ar putea efectua o grupare cu ajutorul unui tip de enumerare, astfel:

```
enum FileException { Open, Source, Target, Temp };
```

Apoi, în program, erorile se pot lansa astfel:

```
throw FileException(Source);
```

Când programul preia excepția, se poate detecta valoarea corespunzătoare folosind o instrucțiune *switch*. Următorul program, COMBINE.CPP, folosește o instrucțiune *switch* pentru a determina excepția lansată:

```

#include <iostream.h>
#include <stdlib.h>

enum FileException { Open, Source, Target, Temp };

void some_operation(void)
{
    throw FileException(1);
}

void main(void)
{
    try {
        some_operation();
    }
    catch (FileException value) {
        switch (value) {
            case Open: cerr << "Open file error" << endl;
                       break;
            case Source: cerr << "Source file error" << endl;
                        break;
            case Target: cerr << "Target file error" << endl;
                        break;
            case Temp: cerr << "Temporary file error" << endl;
                      break;
        };
        exit(1);
    }
}

```

După cum se observă, programul preia excepția și creează o variabilă denumită *value*. Folosind o instrucțiune SWITCH, programul poate determina tipul excepției survenite. Exersați cu acest program schimbând valoarea excepției lansate și urmăriți cum programul își schimbă rezultatele.

### CE SE ÎNTÂMPLĂ LA LANSAREA UNEI EXCEPȚII

Când o funcție generează o excepție, ea nu își mai continuă execuția, ca și când s-ar fi executat o instrucțiune *return*. Apoi, lanțul de apel al funcțiilor este întrerupt până la găsirea unei rutine de tratare a acelei excepții. Într-o rutină de tratare, activată printr-o instrucțiune *catch*, execuția continuă cu acea rutină. De

## Succes cu C++

exemplu, următorul program, EXPCHAIN.CPP, apelează o serie de funcții până când funcția *three* lansează o excepție. Excepția este preluată de *main*:

```
#include <iostream.h>

class Exception { };

void three(void)
{
    cout << "In three, about to throw exception" << endl;
    throw Exception();
}

void two(void)
{
    cout << "In two, about to call three" << endl;
    three();
    cout << "Back in two" << endl;
}

void one(void)
{
    cout << "In one, about to call two" << endl;
    two();
    cout << "Back in one" << endl;
}

void main(void)
{
    try {
        one();
    }
    catch (Exception) {
        cout << "Caught the exception in main" << endl;
    }

    cout << "Hello, world" << endl;
}
```

Când C++ întrerupe lanțul de apeluri în căutarea unei funcții ce tratează excepția, instrucțiunile din funcțiile întrerupte nu mai sunt executate. De exemplu, dacă compilați și executați acest program, pe ecran va apărea:

```
C:\> EXPCHAIN <ENTER>
In one, about to call two
In two, about to call three
In three, about to throw exception
Caught the exception in main
Hello, world
```

După cum se observă, câteva instrucțiuni din fiecare din funcțiile anterioare nu se mai execută. Exersați cu acest program, incluzând o rutină de tratare în funcția *two*, astfel:

```
void two(void)
{
    cout << "In two, about to call three" << endl;
    try {
        three();
    }
    catch (Exception) {
        cout << "Caught the exception in two" << endl;
    }
    cout << "Back in two" << endl;
}
```

La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> EXPCHAIN <ENTER>
In one, about to call two
In two, about to call three
In three, about to throw exception
Caught the exception in two
Back in two
Back in one
Hello, world
```

Când excepția este preluată de funcția *two*, aceasta își va continua execuția, va ceda controlul funcției *one*, care, la rândul ei, va ceda controlul funcției *main*. Deoarece excepția a fost tratată de funcția *two*, ea nu va mai fi activă când controlul revine funcției *main*, de aceea va fi executată instrucțiunea *catch*.



#### EXCEPȚIILE ÎNTRERUP LANȚUL DE APEL AL FUNCȚIILOR

Când o funcție lansează o excepție, execuția funcției este oprită, ca și cum funcția ar fi executat o instrucțiune *return*. C++ caută apoi în lanțul de apel o funcție de tratare a excepției. Dacă funcția următoare din lanțul de apel nu tratează excepția, execuția acelei funcții este imediat oprită. Dacă o funcție a creat un obiect local, este apelată funcția destructor a obiectului înainte ca funcția să își încheie execuția.

#### ATENȚIE LA EXCEPȚIILE SUBPROGRAMELOR DE BIBLIOTECĂ

Pe măsură ce folosirea excepțiilor ia amploare, veți întâlni funcții de bibliotecă și funcții executate la rulare ce generează excepții. Când citiți documentația

acestor funcții, notați, posibilele excepții pe care acestea le pot lansa. Dacă programul nu preia astfel de excepții, atunci va fi apelată rutina de tratare prestabilită, care va încheia execuția programului în majoritatea situațiilor.

### SPECIFICAREA INTERFEȚEI PENTRU EXCEPȚII

Așa cum se știe, prototipul unei funcții specifică tipul de valori returnat de funcție și tipul parametrilor. Când aveți funcții ce generează excepții, se pot specifica posibilele excepții printr-o instrucțiune similară unui prototip de funcție. De exemplu, să presupunem că funcția *perform\_copy* poate genera excepțiile *open\_error*, *read\_error* și *write\_error*. Prototipul funcției va putea fi specificat astfel:

```
void perform_copy(char *source_file, char *target_file)
    throw (open_error, read_error, write_error);
```

La specificarea în acest mod a excepțiilor unei funcții, compilatorul C++ va asocia automat orice eroare nespecificată în listă cu funcția *unexpected*.

### EXCEPȚII PE MAI MULTE NIVELURI

Când programele generează și preiau excepții, apar situații când o rutină de tratare realizează operații care necesită, la rândul lor, tratarea erorilor. Să considerăm, de exemplu, următorul fragment de program. Instrucțiunile activează tratarea erorilor folosind instrucțiunea *try*. Apoi, instrucțiunile apelează funcția *some\_operation*. Dacă operația lansează excepția *out\_of\_memory*, rutina de tratare va prelua excepția și va derula instrucțiuni de tratare a unor excepții suplimentare:

```
try {
    some_operation();
}
catch (out_of_memory) {
    // statements

    try {
        some_other_operation();
    }

    catch (cant_write_to_file) {
        // statements
    }
}
```

În funcție de complexitatea programului, pot exista mai multe niveluri de excepții incluse unele în altele.

### REPETAREA UNEI OPERAȚII

Pentru simplitate, majoritatea programelor precedente se încheie la întâlnirea unei excepții. În multe situații, însă, se dorește rezolvarea cauzei erorii pentru a putea repeta operația ce a lansat eroarea. La începutul acestui capitol, programul *ARRAYEXEC.CPP* descoperea valori eronate de indici transmise funcțiilor *fill\_array* și *show\_array*. Următorul program, *TRYAGAIN.CPP*, preia astfel de indici eronați și decrementează continuu valoarea acestora, până când este găsită o valoare acceptabilă de index:

```
#include <iostream.h>
#include <stdlib.h>

class Array {
public:
    Array(int size);
    void fill_array(int num_values);
    void show_array(int num_values);
    class range { };
private:
    int *buffer;
    int size;
};

Array::Array(int size)
{
    buffer = new int[size];
    Array::size = size;
}

void Array::fill_array(int num_values)
{
    if (num_values > size)
        throw range();

    for (int i = 0; i < num_values; i++)
        buffer[i] = i;
}

void Array::show_array(int num_values)
{
    if (num_values > size)
        throw range();

    for (int i = 0; i < num_values; i++)
        cout << buffer[i] << endl;
}

void main(void)
```

```

{
    Array vector(200);
    int index = 500;

    fill:
    try {
        vector.fill_array(index);
    }
    catch (Array::range) {
        index -= 1;
        goto fill;
    }

    index = 500;

    show:
    try {
        vector.show_array(index);
    }
    catch (Array::range) {
        index -= 1;
        goto show;
    }
}

```

De fiecare dată când o excepție este preluată pentru tratare, programul scade o unitate din indicele incorect, apoi folosește instrucțiunea *goto* pentru a se conecta la o locație ce precede instrucțiunea *try*, în scopul repetării operației. Programul trebuie să transfere controlul înainte de instrucțiunea *try* pentru a permite tratarea excepțiilor pentru următorul apel de funcție. Exersați cu acest program și apeleți funcția din interiorul rutinei de tratare, cum se arată mai jos:

```

try {
    vector.fill_array(index);
}
catch (Array::range) {
    index -= 1;
    vector.fill_array(index);
}

```

Deoarece tratarea erorilor nu a fost reluată, programul va apela rutina prestabilită de tratare, după care se va încheia.

### TRATAREA EXCEPȚIILOR NEPRELuate

Așa cum ați învățat, când o excepție este lansată, dar omisă, se va executa o rutină de tratare prestabilită, care va încheia execuția programului. Numele acestei rutine implicite este *terminate*. Următorul program, *USE\_TERM.CPP*,

apelează pur și simplu funcția *terminate* pentru a verifica existența acesteia. Observați că programul include fișierul antet *EXCEPT.H*. În funcție de compilatorul pe care îl aveți la dispoziție, numele fișierului antet ce conține prototipul funcției *terminate* poate fi ușor diferit:

```

#include <iostream.h>
#include <except.h>

```

```

void main(void)
{
    terminate();
}

```

În mod prestabilit, funcția *terminate* afișează un mesaj și apoi încheie execuția. În alte situații s-ar putea crea o funcție proprie care să înlocuiască funcția *terminate*. Acest lucru se poate realiza cu ajutorul funcției *set\_terminate*. Următorul program, *HASTA.CPP*, folosește funcția *set\_terminate* pentru a apela funcția *my\_termination* în cazul unei excepții generate, dar omise:

```

#include <iostream.h>
#include <except.h>
#include <stdlib.h>

```

```

class my_error { };
class your_error { };

```

```

void my_termination(void)
{
    cerr << "Hasta la vista, baby!" << endl;
    exit(1);
}

```

```

void some_operation(void)
{
    throw your_error();
}

```

```

void main(void)
{
    set_terminate(my_termination);
    try {
        some_operation();
    }
    catch (my_error) {
        cerr << "Caught my error !" << endl;
    }
}

```

În programul anterior, deoarece excepția *your\_error* este lansată, dar omisă, este apelată funcția *terminate*, care este substituită de funcția *my\_termination*. Aceasta afișează un mesaj și încheie execuția programului prin funcția *exit*.



Așa cum ați învățat, dacă se specifică excepțiile ce pot fi lansate de o funcție, dar această funcție va genera ulterior o excepție nespecificată, programul va apela o rutină prestabilită de tratare denumită *unexpected*. Următorul program, UNEXPECT.CPP, invocă funcția *unexpected*, pentru a verifica existența acesteia:

```
#include <iostream.h>
#include <except.h>
```

```
void main(void)
{
    unexpected();
}
```

Există situații la generarea unei excepții nespecificate când se dorește apelarea altei funcții, în locul funcției *unexpected*. Această substituție se poate face cu ajutorul funcției *set\_unexpected*. Programul următor, SET\_UNEX.CPP, folosește funcția *set\_unexpected* pentru a instala o rutină proprie de tratare, care se execută când o funcție lansează o excepție nespecificată:

```
#include <iostream.h>
#include <except.h>
#include <stdlib.h>
```

```
class my_error { };
class your_error { };
```

```
void my_unexpected_handler(void)
{
    cerr << "Shame on someone for throwing an unlisted exception!" << endl;
    exit(1);
}
```

```
void some_operation(void) throw (my_error)
{
    throw your_error();
}
```

```
void main(void)
{
    set_unexpected(my_unexpected_handler);
    try {
        some_operation();
    }
    catch (my_error) {
        cerr << "Caught my error !" << endl;
    }
}
```

## REZUMAT

Tratarea excepțiilor permite programelor să preia și să trateze o gamă largă de erori definite de program. Dacă compilarea programelor prezentate în acest capitol nu reușește, încercați să procurați o versiune mai nouă a compilatorului C++. În Capitolul 15 veți învăța cum să controlați și să gestionați zona de memorie alocabilă (heap). Înainte de a trece la Capitolul 15, verificați dacă ați învățat următoarele:

- ✓ În cel mai simplu sens, o excepție este o eroare definită de program. O rutină de tratare a excepției (handler) este o secvență de program scrisă pentru a răspunde unei astfel de erori.
- ✓ În C++, tratarea erorilor este realizată de instrucțiunile *try*, *throw* și *catch*.
- ✓ Instrucțiunea *try* se folosește într-un program pentru a valida tratarea excepțiilor.
- ✓ Pentru a determina apariția unei excepții se folosește instrucțiunea *catch*.
- ✓ Pentru a determina apariția unei anumite excepții se folosește instrucțiunea *catch*.
- ✓ Pentru generarea unei excepții, se folosește instrucțiunea *throw*. Cu alte cuvinte, când apare o eroare, programele lansează o excepție ce va fi ulterior preluată de rutina de tratare corespunzătoare.
- ✓ Fiecare excepție definită de program are un nume unic. De obicei, se folosește câte o clasă pentru fiecare excepție. Prin utilizarea unei clase, programele pot atribui valori variabilelor membru ale clasei, pentru a furniza mai multe informații rutinei de tratare a excepției respective.
- ✓ Dacă o excepție este lansată, dar omisă, se va executa o rutină de tratare prestabilită, denumită *terminate*. În majoritatea situațiilor, funcția *terminate* afișează un mesaj de eroare și oprește execuția programului. Programele își pot defini propriile rutine de încheiere folosind funcția *set\_terminate*. Fișierul antet EXCEPT.H conține prototipul funcției *set\_terminate*.
- ✓ Așa cum programele folosesc prototipuri de funcții pentru specificarea tipului valorii unei funcții și a parametrilor ei, la fel poate fi specificată o listă de excepții pe care le poate lansa o funcție. Dacă o funcție generează o excepție ce nu este conținută în listă, C++ va apela o rutină specială de tratare, denumită *unexpected*. Folosind funcția *set\_unexpected*, programele își pot defini propriile rutine de tratare a excepțiilor nespecificate. Fișierul antet EXCEPT.H conține prototipul funcției *set\_unexpected*, în funcție de compilatorul folosit.

## CAPITOLUL 15

### APROFUNDAREA GESTIUNII ZONEI DE MEMORIE ALOCABILĂ

În Capitolul 7 ați învățat cum se alocă și se eliberează dinamic memoria în C++, folosind operatorii *new* și *delete*. Ați învățat că atunci când realizați alocarea dinamică a memoriei, operatorul *new* obține memorie dintr-o zonă denumită *heap* (zonă de memorie alocabilă). Zona alocabilă este, așadar, un ansamblu de locații de memorie aflate la dispoziția programului. Pe măsură ce crește complexitatea programelor ce realizează alocarea dinamică a memoriei, apar situații când trebuie să examinați diferite proprietăți ale zonei alocabile, pentru a putea depista erorile dificile din program. Dacă lucrați într-un mediu DOS, programele pot folosi funcțiile prezentate în acest capitol pentru examinarea zonei alocabile. Dacă lucrați în alte medii, rețineți conceptele prezentate în acest capitol, deoarece există funcții similare și în alte medii de lucru.

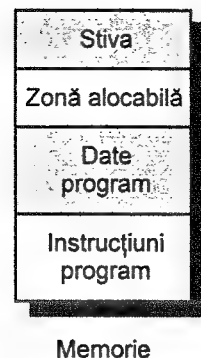
Când veți termina acest capitol, veți cunoaște:

- ♦ Ce este zona alocabilă și care este dimensiunea ei
- ♦ Cum se gestionează regiunile de memorie alocate și libere
- ♦ Cum se pot detecta erorile de alocare a memoriei
- ♦ Cum se poate detecta alterarea zonei alocabile cauzată de erori de programare

**Observație:** Compilatorul pe care îl folosiți poate utiliza nume de funcții diferite de cele din acest capitol, care se referă îndeosebi la compilatorul Borland C++.

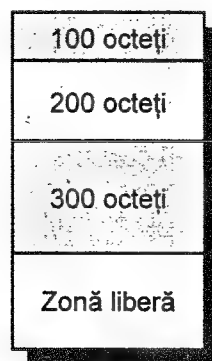
#### EXAMINAREA ZONEI DE MEMORIE ALOCABILĂ

Zona alocabilă este o regiune de memorie din care programele pot alocă dinamic porțiuni de memorie, cum se arată în Figura 15.1.



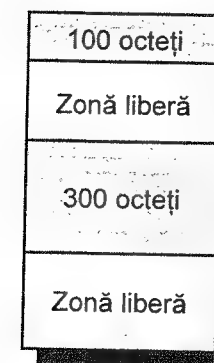
**Figura 15.1** Zonă alocabilă (heap) este o porțiune a memoriei din care programele pot alocă memorie în mod dinamic.

Presupunând că un program alocă trei regiuni în memorie cu dimensiunile de 100, 200 și respectiv 300 de octeți, zona alocabilă va arăta ca în Figura 15.2.



**Figura 15.2** Alocarea memoriei în zona alocabilă.

Dacă ulterior programul va elibera 200 de octeți, zona alocabilă va arăta ca în Figura 15.3.



**Figura 15.3.** Eliberarea memoriei în zona alocabilă.

Pentru a permite alocarea dinamică a memoriei, sistemul de operare sau limbajul de programare furnizează funcții de administrare a memoriei, a căror activitate este invizibilă operatorului uman. Pentru a alocă și a elibera memorie din zona alocabilă, aceste funcții urmăresc fiecare regiune care a fost alocată, precum și pe cele disponibile. În cazul sistemului de operare DOS, majoritatea compilatoarelor creează o listă cu legături bazată pe structura *heapinfo* arătată în continuare:

```
struct heapinfo {
    void *ptr;
    unsigned int size;
    int in_use;
};
```

Lista cu legături conține un nod *heapinfo* pentru fiecare regiune de memorie liberă sau folosită. Pointerul *ptr* indică următorul nod din lista înlanțuită. Componenta *size* specifică dimensiunea (în octeți) a regiunii de memorie asociată nodului curent. Componenta *in\_use* primește valoarea 1 dacă regiunea de memorie este alocată și 0 dacă este disponibilă. Dată fiind alocarea anterioară de 100, 200 și 300 de octeți în zona alocabilă, lista cu legături va arăta ca în Figura 15.4.

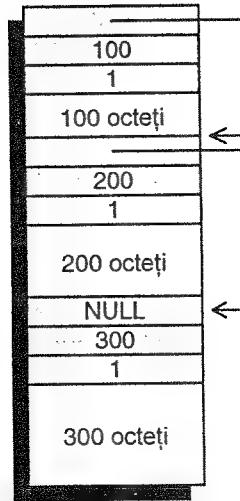


Figura 15.4 Spațiul zonei alocabile este gestionat folosind o listă cu legături.

Dacă programul ar urma să elibereze regiunea de 200 de octeți, valoarea componentei *in\_use* a celui de al doilea nod se va schimba din 1 (folosit) la 0 (disponibil).

Pentru a vă ajuta să înțelegeți mai bine lista cu legături a zonei alocabile, următorul program, HEAPLIST.CPP, alocă memorie și apoi urmărește lista cu legături a zonei alocabile folosind funcția *heapwalk*. Programul eliberează apoi regiunile alocate și afișează noua listă cu legături:

```
#include <iostream.h>
#include <alloc.h>

void main(void)
{
    char *buffer1, *buffer2, *buffer3, *buffer4;

    struct heapinfo node = {NULL, 0, 0};

    buffer1 = new char[1000];
    buffer2 = new char[2000];
    buffer3 = new char[3000];
    buffer4 = new char[4000];

    while (heapwalk(&node) == _HEAPOK)
    {
        cout << "Size " << node.size << " bytes State ";
```

```
        if (node.in_use)
            cout << "In use" << endl;
        else
            cout << "Free" << endl;
    }

    node.ptr = NULL; // Start at the beginning at the list

    delete buffer2;
    delete buffer4;

    cout << endl << "After memory release" << endl;

    while (heapwalk(&node) == _HEAPOK)
    {
        cout << "Size " << node.size << " bytes State ";

        if (node.in_use)
            cout << "In use" << endl;
        else
            cout << "Free" << endl;
    }
}
```


La compilarea și execuția aceluși program, pe ecran se vor afișa următoarele rezultate, care s-ar putea să vă deruteze:

```
C:\> HEAPLIST <ENTER>
Size 516 bytes State In use
Size 40 bytes State In use
Size 520 bytes State In use
Size 40 bytes State In use
Size 520 bytes State In use
Size 40 bytes State In use
Size 520 bytes State In use
Size 14 bytes State In use
Size 16 bytes State In use
Size 8 bytes State In use
Size 1004 bytes State In use
Size 2004 bytes State In use
Size 3004 bytes State In use
Size 4004 bytes State In use

After memory release
Size 516 bytes State In use
Size 40 bytes State In use
Size 520 bytes State In use
Size 40 bytes State In use
Size 520 bytes State In use
Size 40 bytes State In use
```

Size 520 bytes State In use  
 Size 14 bytes State In use  
 Size 16 bytes State In use  
 Size 8 bytes State In use  
 Size 1004 bytes State In use  
 Size 2004 bytes State Free  
 Size 3004 bytes State In use

Așa cum se poate vedea, zona alocabilă conține intrări și pentru regiuni de memorie diferite de bufferele alocate de program. În funcție de compilatorul folosit, motivul acestor alocări ar putea să fie diferit. De exemplu, alocarea ar putea fi efectuată automat pentru buffer-ele fluxurilor I/O. De asemenea, să observăm că în acest exemplu, nodul asociat fiecărei regiuni necesită 4 octeți pentru regiuni de memorie *near* (care folosesc pointeri pe 16 biți) și 8 octeți pentru regiuni de memorie *far* (care folosesc pointeri pe 32 biți). Dacă folosiți alt sistem de operare decât MS-DOS, compilați acest program folosind modelele de memorie *small* și *large*. În cazul modelului de memorie *small*, nodul va folosi pointeri pe 16 biți, iar în cazul modelului de memorie *large*, pe 32 de biți.



**EVIDENȚA LISTEI CU LEGĂTURI A ZONEI ALOCABILE**

Zona alocabilă este o parte a memoriei pe care programele o alocă dinamic. Pe măsură ce programele alocă și eliberează memorie, zona alocabilă ar putea conține diferite porțiuni, unele în folosință iar altele libere. Pentru a controla regiunile zonei alocabile, majoritatea limbajelor de programare folosesc o listă cu legături ale cărei noduri corespund regiunilor zonei. Pentru a putea detecta erorile ce apar la programele care folosesc alocarea dinamică a memoriei, bibliotecile de subprograme furnizează un set de funcții ce pot fi folosite la parcurgerea listei înlanțuite a zonei alocabile. Prin examinarea individuală a conținutului zonei alocabile, se pot detecta multe erori de programare.

### TESTAREA VALIDITĂȚII UNEI INTRĂRI DIN ZONA ALOCABILĂ

La alocarea memoriei în zona alocabilă, pot apărea situații când, datorită unei erori de programare, zona alocabilă este deteriorată prin suprascrierea unor date ale programului. Să presupunem, de exemplu, că un program alocă un buffer de 50 de octeți și alt buffer de 100 de octeți, cum se arată în Figura 15.5.

## 15: Gestiunea zonei de memorie alocabilă

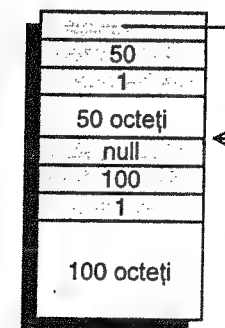


Figura 15.5 Alocarea a două regiuni de memorie în zona alocabilă.

Să presupunem, apoi, că programul folosește un ciclu *for* pentru a atribui valoarea 0 celor 50 de elemente ale primului buffer;

```
for (i = 0; i <= 50; i++)
    fifty[i] = 0;
```

Deoarece ciclul se repetă până când variabila *i* este mai mică sau egală cu 50, valoarea 0 va fi atribuită și unei locații situată în afara bufferului. După cum se vede în Figura 15.6, această atribuire suprascrie un octet din componenta pointer a nodului asociat celei de a doua intrări în zona alocabilă.

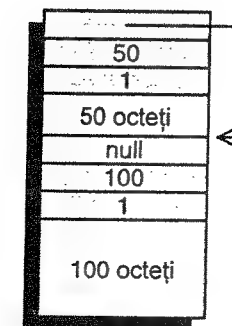


Figura 15.6 Suprascrierea unui octet în pointerul altui nod datorată unei erori de programare similare.

Următorul program, OVERWRIT.CPP, ilustrează o eroare de programare:

```
#include <iostream.h>
#include <string.h>
void main(void)
```

```
{
char *fifty = new char[50];
char *string = new char[100];

strcpy(string, "Success with C++");

for (int i = 0; i <= 60; i++)
    fifty[i] = 0;

cout << string << endl;
}
```

Se observă că programul alocă două buffere de memorie și atribuie celui de al doilea buffer un șir de caractere. Apoi, programul folosește un ciclu *for* pentru a atribui valoarea 0 fiecărui octet din primul buffer. După cum se vede în Figura 15.7, programul suprascrie o porțiune a celui de al doilea buffer:

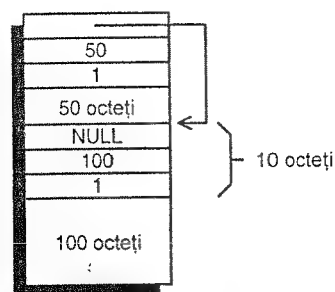



Figura 15.7 Suprascrierea unei regiuni a zonei alocabile.

Deoarece programul anterior este scurt, cauza erorii se poate detecta cu ușurință, folosind tehnicile de depanare obișnuite. Pe măsură, însă, ce programele devin mai complicate, este nevoie de a examina periodic intrările zonei alocabile, în scopul determinării locului apariției erorilor. Următoarele secțiuni prezintă câteva tehnici de programare folosite pentru a testa starea zonei alocabile.



**SĂ ÎNȚELEM STAREA ZONEI ALOCABILE**

Pentru a controla memoria din zona alocabilă, majoritatea limbajelor de programare folosesc o listă cu legături. Din nefericire, dacă un program conține erori, el poate să suprascrie octeți aparținând nodurilor listei. În astfel de situații, zona alocabilă este deteriorată. Pentru a determina starea zonei alocabile, multe din bibliotecile de subprograme oferă funcții ce ajută la examinarea tuturor intrărilor zonei alocabile, sau a anumitor intrări. În acest mod, programul poate să verifice dacă anumite intrări ale zonei sunt incorecte.

### O VERIFICARE RAPIDĂ A ZONEI ALOCABILE

În funcție de erorile care trebuie detectate, se poate dori, la un moment dat, verificarea validității intrărilor zonei alocabile. În astfel de situații, programele pot folosi funcția de bibliotecă *heapcheck*. Această funcție returnează una din valorile indicate în Tabela 15.1.

Valoare returnată	Explicația
_HEAPEMPTY	Nu există zonă alocabilă
_HEAPOK	Intrările în zonă sunt valide
_HEAPCORRUPT	Una sau mai multe intrări sunt deteriorate

Tabela 15.1 Valorile de stare ale zonei alocabile returnate de funcția *heapcheck*.

Următorul program, *HEAPCHK.CPP*, folosește funcția *heapcheck* pentru a determina dacă zona alocabilă a fost deteriorată:

```
#include <iostream.h>
#include <string.h>
#include <alloc.h>

void main(void)
{
    char *fifty = new char[50];
    char *string = new char[100];

    strcpy(string, "Success with C++");

    switch (heapcheck()) {
        case _HEAPEMPTY:    cout << "No heap" << endl;
                           break;
        case _HEAPOK:      cout << "Heap is valid" << endl;
                           break;
        case _HEAPCORRUPT: cout << "Heap is corrupted" << endl;
                           break;
    };

    for (int i = 0; i <= 60; i++)
        fifty[i] = 0;

    switch (heapcheck()) {
        case _HEAPEMPTY:    cout << "No heap" << endl;
                           break;
        case _HEAPOK:      cout << "Heap is valid" << endl;
                           break;
        case _HEAPCORRUPT: cout << "Heap is corrupted" << endl;
                           break;
    };
}
```

```
};

cout << string << endl;
}
```

După cum se observă, programul folosește funcția *heapcheck* înainte și după deteriorarea zonei alocabile de către ciclul *for*. La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> HEAPCHK <ENTER>
Heap is valid
Heap is corrupted
```

Folosind funcția *heapcheck* în acest mod se poate detecta locul exact din program unde zona alocabilă se alterează.

### VALIDAREA UNEI ANUMITE INTRĂRI DIN ZONA ALOCABILĂ

La depanarea intrărilor deteriorate ale zonei alocabile, uneori este necesară examinarea de către program a fiecărui pointer al nodurilor listei. Pentru a determina validitatea intrărilor în zonă, programele pot folosi funcția *heapchecknode*. Funcția returnează una din valorile prezentate în Tabela 15.2.

Valoarea returnată	Explicația
_HEAPEMPTY	Nu există zonă alocabilă
_HEAPOK	Intrările în zonă sunt valide
_HEAPCORRUPT	Una sau mai multe intrări sunt deteriorate
_BADNODE	Nodul curent este deteriorat
_FREENTRY	Blocul este liber
_USEDENTRY	Blocul este valid și este folosit

Tabela 15.2 Valorile de stare ale zonei alocabile returnate de funcția *heapchecknode*.

Următorul program, *NODECHK.CPP*, folosește funcția *heapchecknode* pentru a examina pointerii programului:

```
#include <iostream.h>
#include <string.h>
#include <alloc.h>

void main(void)
{
    char *fifty = new char[50];
    char *string = new char[100];
```

```
strcpy(string, "Success with C++");

for (int i = 0; i <= 60; i++)
    fifty[i] = 0;

switch (heapchecknode(fifty)) {
    case _HEAPEMPTY:    cout << "No heap" << endl;
                        break;
    case _HEAPOK:       cout << "Heap is valid" << endl;
                        break;
    case _HEAPCORRUPT:  cout << "Heap is corrupted" << endl;
                        break;
    case _BADNODE:      cout << "Node is corrupted" << endl;
                        break;
    case _FREENTRY:     cout << "Entry is not in use" << endl;
                        break;
    case _USEDENTRY:    cout << "Entry is valid and in use" << endl;
                        break;
}

switch (heapchecknode(string)) {
    case _HEAPEMPTY:    cout << "No heap" << endl;
                        break;
    case _HEAPOK:       cout << "Heap is valid" << endl;
                        break;
    case _HEAPCORRUPT:  cout << "Heap is corrupted" << endl;
                        break;
    case _BADNODE:      cout << "Node is corrupted" << endl;
                        break;
    case _FREENTRY:     cout << "Entry is not in use" << endl;
                        break;
    case _USEDENTRY:    cout << "Entry is valid and in use" << endl;
                        break;
}

cout << string << endl;
}
```

Se observă că programul folosește funcția *heapchecknode* pentru fiecare variabilă pointer. La compilarea și execuția programului, pe ecran se va afișa:

```
C:\> NODECHK <ENTER>
Entry is valid and in use
Heap is corrupted
```

### TESTAREA SPAȚIULUI DE MEMORIE DISPONIBIL DIN ZONA ALOCABILĂ

La alocarea și eliberarea, în timp, a memoriei, spațiul disponibil poate deveni fragmentat, zona alocabilă fiind formată din spații libere și spații alocate dispuse alternativ, cum se arată în Figura 15.8.

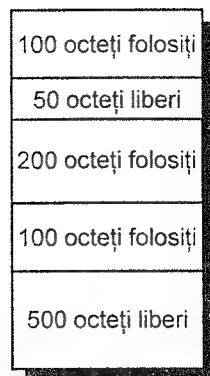


Figura 15.8 Spațiul liber fragmentat din memoria alocabilă.

Dacă un program suprascrie o porțiune de memorie disponibilă din zona alocabilă, efectul erorii poate să apară mai târziu, când programul va încerca să aloce zona de memorie disponibilă. Pentru a detecta astfel de erori, se pot folosi funcțiile *heapfillfree* și *heapcheckfree*. Funcția *heapfillfree* atribuie o valoare specifică tuturor locațiilor spațiului liber. De exemplu, programul ar putea atribui valoarea -1 fiecărei regiuni de memorie disponibilă. Ulterior, funcția *heapcheckfree* examinează spațiul liber al zonei alocabile pentru a se asigura că fiecare locație conține valoarea specificată. Dacă programul suprascrie o locație din memoria disponibilă, funcția *heapcheckfree* va detecta schimbarea, permițând programului să detecteze eroarea. Funcția *heapcheckfree* returnează una din valorile prezentate în Tabela 15.3.

Valoarea returnată	Explicația
_HEAPEMPTY	Nu există zonă alocabilă
_HEAPOK	Intrările din zonă sunt corecte
_HEAPCORRUPT	Una sau mai multe intrări sunt deteriorate
_BADVALUE	O altă valoare a fost întâlnită într-o locație liberă

Tabela 15.3 Valorile de stare ale zonei alocabile returnate de funcția *heapcheckfree*.

Următorul program, *CHKFREE.CPP*, folosește funcția *heapfillfree* pentru a atribui valoarea -1 spațiului liber de memorie. Programul folosește apoi un ciclu *for* eronat pentru a suprascrie un buffer (din spațiul liber). La terminarea ciclului *for* programul detectează eroarea folosind funcția *heapcheckfree*:

```
#include <iostream.h>
#include <alloc.h>
```

```
void main(void)
{
    char *start, *middle, *end;
    int state;

    start = new char[50];
    middle = new char[50];
    end = new char[50];

    delete middle;    // Create free space in the middle

    state = heapfillfree(-1);

    if (state == _HEAPOK)
        cout << "Heap is ok" << endl;
    else if (state == _HEAPCORRUPT)
        cout << "Heap is corrupt" << endl;

    for (int i = 0; i <= 60; i++)
        start[i] = i;

    state = heapcheckfree('A');

    if (state == _HEAPOK)
        cout << "Heap is ok" << endl;
    else if (state == _HEAPCORRUPT)
        cout << "Heap is corrupt" << endl;
    else if (state == _BADVALUE)
        cout << "Value has been changed in free space" << endl;
}
```

După cum se observă, programul alocă trei buffere de memorie, apoi eliberează bufferul din mijloc pentru a crea spațiu liber. La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> CHKFREE <ENTER>
Heap is ok
Value has been changed in free space
```

### COMPACTAREA SPAȚIULUI DE MEMORIE DISPONIBIL

Cu timpul, spațiul de memorie liber și cel alocat pot deveni alocabile. Să presupunem, de exemplu, că zona alocabilă apare ca în Figura 15.8. Dacă programul eliberează regiunea de 200 de octeți, funcțiile de administrare a zonei vor compacta cele două regiuni în una singură, cum se arată în Figura 15.9.



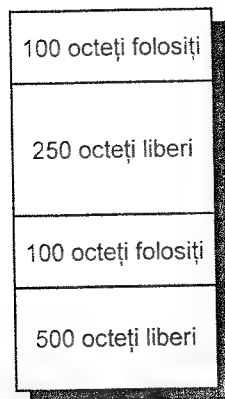


Figura 15.9 Compactarea spațiului disponibil din zona alocabilă.

Așa cum se vede în Figura 15.9, zona alocabilă conține 750 de octeți de spațiu disponibil. Deoarece spațiul este fragmentat, cea mai mare porțiune de memorie pe care programul o poate alocă este de 500 de octeți. Din păcate compilatorul C++ nu realizează *colectarea resturilor*, care ar duce, periodic, la unificarea tuturor spațiilor libere, cum se arată în Figura 15.10.

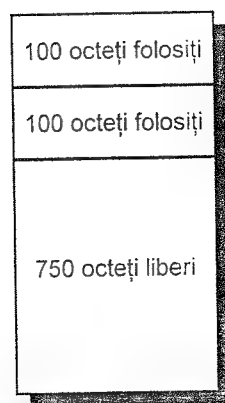


Figura 15.10 Colectarea resturilor este procesul de compactare a spațiului disponibil din zona alocabilă.

Deoarece programele conțin pointeri la anumite regiuni de memorie, rutinele de administrare a zonei alocabile nu pot deplasa regiunile de memorie pentru a

## 15: Gestiunea zonei de memorie alocabilă

compacta spațiul disponibil (rutinele de gestiune nu au cunoștință de variabilele cărora li s-au atribuit valorile pointerilor, iar dacă memoria este deplasată pointerii vor fi eronați). Dacă un program nu poate alocă spațiu datorită fragmentării, atunci programele vor elibera și realoca memorie succesiv, realizând astfel propria colectare de resturi.

### REZUMAT

Când programele alocă și eliberează memorie în mod dinamic, folosind operatorii *new* și *delete*, pot apărea erori greu de detectat. Pentru a contribui la depănarea unor astfel de erori, majoritatea compilatoarelor sau sistemelor de operare furnizează un set de funcții de bibliotecă ce permit testarea stării zonei alocabile. Folosiți aceste funcții ori de câte ori aveți neazuri cu alocarea dinamică a memoriei. Înainte de a trece la Capitolul 16, asigurați-vă că ați învățat următoarele:

- ✓ Zona alocabilă este o regiune de memorie pe care programele o alocă dinamic. În funcție de sistemul de operare sau de modelul de memorie a compilatorului, dimensiunea acestei zone variază.
- ✓ Pentru gestiunea spațiilor de memorie disponibile sau alocate, se folosește o listă cu legături. Pentru fiecare intrare, zona alocabilă controlează dimensiunea și disponibilitatea acesteia (dacă zona este sau nu disponibilă).
- ✓ Pentru a detecta o eroare de alocare a memoriei, programele pot parcurge intrările listei zonei alocabile.
- ✓ Prin examinarea intrărilor zonei alocabile utilizând acest procedeu, programele pot detecta deteriorările locațiilor de memorie cauzate de erori de programare.



## CAPITOLUL 16

### SĂ CONSTRUIM CLASE CU OPERATORI I/O INTRINSECI

Multe din programele din această carte au folosit funcții membru pentru afișarea variabilelor membru ale unei clase, cum au fost *show\_class*, *show\_book*, *show\_employee*. Urmărind acest model, programele ar putea folosi funcții membru ca *file\_class* sau *file\_employee* pentru a transcrie membrii unei clase într-un stream fișier. Când proiectați programe, acestea vor deveni mai ușor de înțeles prin scrierea unor clase cu operatori I/O intrinseci.

În cel mai simplu sens, o clasă cu operatori I/O intrinseci este o clasă ce suportă operatorii de inserție (<<) și extracție (>>). Folosind acești operatori I/O, programele pot ușor realiza operații I/O cu tastatura, ecranul sau fișierele, așa cum se folosesc stream-urile I/O *cin* și *cout*. Acest capitol examinează în detaliu clasele cu operatori I/O intrinseci. În particular, veți învăța:

- ♦ Ce este și cum se creează o clasă cu operatori I/O intrinseci
- ♦ Cerințele legate de operatorii suprapuși
- ♦ Cum suprapunerea operatorilor pentru crearea claselor cu operatori I/O intrinseci este similară creării unui manipulator
- ♦ Unde se declară funcția apelată de operatorul supradefinit
- ♦ Ce tip are rezultatul și primul parametru al funcției apelate de operatorul suprapus
- ♦ Cum se definește tipul unei clase cu operatori I/O intrinseci
- ♦ Despre clasele cu operatori I/O intrinseci și operațiile I/O pe fișiere
- ♦ Cum se folosesc operatorii de inserție și extracție

#### INTRĂRI/IEȘIRI CU MEMBRII CLASEI ÎN STIL CLASIC

Majoritatea programelor C++ folosesc funcții membru pentru a realiza operații I/O cu tastatura, ecranul sau fișierele. De exemplu, următorul program, TRAVEL.CPP, folosește clasa *travel*:

## Succes cu C++

```
class travel {
public:
    travel(char *destination, int distance);
    void display_trip(void);
    int file_trip(ofstream& file);
private:
    char destination[64];
    int distance;
};
```

Așa cum se observă, clasa folosește funcțiile membru *display\_trip* și *file\_trip* pentru a efectua operații de ieșire cu ecranul și pe fișiere. Următoarele instrucțiuni implementează TRAVEL.CPP:

```
#include <fstream.h>
#include <string.h>

class travel {
public:
    travel(char *destination, int distance);
    void display_trip(void);
    int file_trip(ofstream& file);
private:
    char destination[64];
    int distance;
};

travel::travel(char *destination, int distance)
{
    strcpy(travel::destination, destination);
    travel::distance = distance;
}

void travel::display_trip(void)
{
    cout << "Destination: " << destination << " Distance: " <<
        distance << endl;
}

int travel::file_trip(ofstream& out_file)
{
    out_file << destination << endl;
    out_file << distance << endl;
    return(1);
}

void main(void)
{
    travel vacation("Maui", 2500);
```

## 16: Clase cu operatori I/O intrinseci

```
vacation.display_trip();

ofstream trip_file("TRAVEL.DAT");

vacation.file_trip(trip_file);
}
```

Programul apelează cele două funcții membru pentru a realiza operații I/O. La compilarea și execuția programului, pe ecran se va afișa:

```
C:\> TRAVEL <ENTER>
Destination: Maui Distance: 2500
```

Este important de observat că majoritatea programelor C++ folosesc funcții membru pentru a realiza operații I/O bazate pe clase. Problema care apare la folosirea acestor funcții membru este aceea că, la citirea programului, trebuie înțelese funcțiile suplimentare.

O clasă cu operatori I/O intrinseci, pe de altă parte, folosește operatorul de inserție pentru a realiza astfel de operații I/O:

```
cout << vacation;

output_file << business_trip;
```

Pentru a realiza operații I/O în acest mod, programele trebuie să suprapună operatorul de inserție (<<) pentru fiecare tip de clasă în parte. De exemplu, următorul program, STREAM.CPP, folosește operatorul de inserție pentru a afișa obiectul *travel* și pentru a scrie informații despre călătorie într-un fișier:

```
#include <fstream.h>
#include <string.h>

class travel {
public:
    travel(char *destination, int distance);
    friend ostream& operator <<(ostream& stream, travel object);
    friend ofstream& operator <<(ofstream& stream, travel object);
private:
    char destination[64];
    int distance;
};

travel::travel(char *destination, int distance)
{
    strcpy(travel::destination, destination);
    travel::distance = distance;
}

ostream& operator <<(ostream& stream, travel object)
{
```

```

stream << "Destination: " << object.destination <<
    " Distance: " << object.distance << endl;

return(stream);
}

ofstream& operator <<(ofstream& stream, travel object)
{
    stream << object.destination << endl;
    stream << object.distance << endl;

    return(stream);
}

void main(void)
{
    travel vacation("Maui", 2500);

    cout << vacation;

    ofstream trip_file("TRAVEL.DAT");

    trip_file << vacation;
}

```

La compilarea și execuția acestui program, pe ecran va apărea:

```

C:\> STREAM <ENTER>
Destination: Maui Distance: 2500

```

Majoritatea operațiilor realizate de acest program au fost discutate în capitolele precedente. Totuși, dacă nu înțelegeți programul, nu vă îngrijorați, deoarece vom discuta detaliat în acest capitol fiecare pas al programului. Deocamdată, însă, să înțelegem că o clasă cu operatori I/O intrinseci este o clasă ce folosește operatorii de inserție și extracție pentru a realiza operații I/O bazate pe stream.

**Observație:** La suprapunerea operatorului de inserție sau extracție pentru o anumită clasă, nu se va schimba destinația operatorului pentru alte clase. Astfel, programul precedent poate folosi operatorul de inserție, de exemplu, pentru a scrie caractere pe ecran, în ciuda faptului că programul a suprapus acest operator pentru clasa *travel*. În acest mod, suprapunerea operatorului de inserție pentru o anumită clasă nu diferă față de suprapunerea operatorului de adunare sau scădere, care va afecta numai anumite tipuri.



## SĂ ÎNȚELEM CLASELE CU OPERATORI I/O INTRINSECI

Majoritatea programelor C++ folosesc funcții membru speciale pentru a realiza I/O operații cu tastatura, ecranul sau fișierele. Din păcate, astfel de funcții introduc alte funcții care trebuie înțelese de către cei care doresc să citească programul. O clasă cu operatori I/O intrinseci, pe de altă parte, furnizează funcții ce acceptă operatorii de inserție (<<) și extracție (>>). Prin crearea unor astfel de clase, intrările/ieșirile de clase ale programului se pot realiza într-un mod mai natural, prin operatorii de inserție și extracție.

### CREAREA UNEI CLASE CU OPERATORI I/O INTRINSECI

Pentru crearea unei astfel de clase, programele suprapun operatorii de inserție (<<) și extracție (>>) pentru anumite stream-uri/O. De exemplu, să presupunem că un program folosește clasa *date* arătată în continuare:

```

class date {
public:
    date(int month, int day, int year);
private:
    int month;
    int day;
    int year;
};

```

Pentru a suprapune operatorul de inserție în vederea unor operații de ieșire pe stream-uri cum este *cout*, trebuie suprapus operatorul << astfel:

```

class date {
public:
    date(int month, int day, int year);
    friend ostream& operator <<(ostream& stream, date date_object);
private:
    int month;
    int day;
    int year;
};

```

După cum se observă, suprapunerea operatorului de inserție este similară creării unui manipulator, așa cum s-a discutat în Capitolul 11. Când se suprapune un operator de inserție sau extracție, se definește pur și simplu o funcție ce se execută când operatorul este folosit cu un anumit tip de date. Tipul valorii returnate și primul parametru al funcției sunt întotdeauna de tip stream:

```

    Tip valoare returnată
    Primul parametru al funcției
ostream& operator <<(ostream& stream, date date_object)
{
    // Statements
}

```

Al doilea parametru al funcției specifică tipul de date pentru care se suprapune operatorul:

```

ostream& operator <<(ostream& stream, date date_object)
{
    // Statements
}

```

Al doilea parametru specifică tipul de date

Să observăm că funcția nu este bazată pe clasă, adică nu este o funcție membru a clasei. Totuși, în cadrul clasei, operatorul trebuie declarat *friend*, cum se arată mai jos:

```

class date {
public:
    date(int month, int day, int year);
    friend ostream& operator <<(ostream& stream, date date_object);
private:
    int month;
    int day;
    int year;
};

```

Specificarea operatorului ca friend

Prin declararea operatorului ca *friend*, el va avea acces la membrii de tip *private* ai clasei. Următorul program, DATE\_STR.CPP folosește operații de ieșire pe stream pentru a afișa membrii clasei *date*:

```

#include <iostream.h>

class date {
public:
    date(int month, int day, int year);
    friend ostream& operator <<(ostream& stream, date date_object);
private:
    int month;
    int day;
    int year;
};

date::date(int month, int day, int year)
{
    date::month = month;
    date::day = day;
    date::year = year;
}

```

```

ostream& operator <<(ostream& stream, date date_object)
{
    switch (date object.month) {
        case 1: stream << "January ";
            break;
        case 2: stream << "February ";
            break;
        case 3: stream << "March ";
            break;
        case 4: stream << "April ";
            break;
        case 5: stream << "May ";
            break;
        case 6: stream << "June ";
            break;
        case 7: stream << "July ";
            break;
        case 8: stream << "August ";
            break;
        case 9: stream << "September ";
            break;
        case 10: stream << "October ";
            break;
        case 11: stream << "November ";
            break;
        case 12: stream << "December ";
            break;
    };

    stream << date object.day << ", " << date object.year << endl;

    return(stream);
}

void main(void)
{
    date birthday(9, 30, 94);

    cout << birthday;
}

```

La compilarea și execuția acestui program, pe ecran va apărea:

```

C:\> DATE_STR <ENTER>
September 30, 94

```



### REGULI PENTRU CREAREA DE CLASE CU OPERATORI I/O INTRINSECI

Pentru a crea o clasă cu operatori I/O intrinseci, programul trebuie să suprapună operatorii de inserție (<<) și extracție (>>) pentru o anumită clasă. Suprapunerea acestor operatori se face prin definirea unei funcții ce este apelată de fiecare dată când operatorul este folosit cu un anumit tip de clasă. Când creai funcția, rețineți următoarele reguli:

- Declarați funcția în afara clasei - funcția operator nu este membru al clasei.
- Primul parametru al funcției, ca și tipul rezultatului, trebuie să fie un stream.
- Al doilea parametru al funcției este tipul clasei.
- Dacă operatorul va avea acces la membrii de tip *private* ai clasei, trebuie să fie declarat *friend* în cadrul clasei.

### Operații de intrare pe stream

Pașii efectuați de un program pentru a realiza operații de intrare pe stream sunt aproape similari celor discutați anterior. Deosebirea este că trebuie suprapus operatorul de extracție (>>). Următoarele instrucțiuni ilustrează modul în care programul suprapune operatorul de extracție pentru clasa *date*:

```
istream& operator >>(istream& stream, date *date_object)
{
    cout << "Please type in mm dd yy: " ;

    stream >> date_object->month;
    stream >> date_object->day;
    stream >> date_object->year;

    return(stream);
}
```

După cum se observă, instrucțiunile respectă aceleași reguli discutate anterior. Următorul program, GET\_DATE.CPP, folosește instrucțiuni de intrare pe stream pentru a introduce componentele unui obiect de tipul *date* - luna, ziua și anul:

```
#include <iostream.h>
```

```
class date {
public:
    date(int month, int day, int year);
    friend ostream& operator <<(ostream& stream, date date_object);
    friend istream& operator >>(istream& stream, date *date_object);
private:
```

```
    int month;
    int day;
    int year;
};

date::date(int month, int day, int year)
{
    date::month = month;
    date::day = day;
    date::year = year;
}

ostream& operator <<(ostream& stream, date date_object)
{
    stream << date_object.month << '/' << date_object.day << '/' <<
        date_object.year << endl;

    return(stream);
}

istream& operator >>(istream& stream, date *date_object)
{
    cout << "Please type in mm dd yy: " ;

    stream >> date_object->month;
    stream >> date_object->day;
    stream >> date_object->year;

    return(stream);
}

void main(void)
{
    date birthday(9, 30, 94);

    cin >> &birthday;

    cout << birthday;
}
```

La execuția programului se va cere introducerea unei date. Programul folosește apoi operatorul de extracție pentru a realiza operația de intrare.

### Operații I/O pe stream fișier

Clasele cu operatori I/O intrinseci nu se rezumă numai la stream-urile *cout* și *cin*. De exemplu, următorul program, FILEDATE.CPP, folosește operații I/O pe stream fișier pentru a scrie date pe fișierul DATES.DAT. Aici programul suprapune operatorul de inserție (<<) pentru obiectele clasei *ofstream*:

```
#include <fstream.h>

class date {
public:
    date(int month, int day, int year):
        friend ofstream& operator <<(ofstream& stream,
            date date_object);
private:
    int month;
    int day;
    int year;
};

date::date(int month, int day, int year)
{
    date::month = month;
    date::day = day;
    date::year = year;
}

ofstream& operator <<(ofstream& stream, date date_object)
{
    stream << date_object.month << ' ' << date_object.day << ' ' <<
        date_object.year << endl;

    return(stream);
}

void main(void)
{
    date birthday(9, 30, 94);
    date christmas(12, 25, 94);
    date new_years(1, 1, 95);

    ofstream datefile("DATES.DAT");

    datefile << birthday;
    datefile << christmas;
    datefile << new_years;
}
```

La compilarea și execuția acestui program se vor scrie trei date pe fișierul DATES.DAT. Într-o manieră similară, următorul program READDATE.CPP suprapune operatorul de extracție pentru a citi date dintr-un fișier până la întâlnirea sfârșitului de fișier. Programul suprapune aici operatorul de extracție pentru obiecte de tipul *ifstream*:

```
#include <fstream.h>

class date {
```

```
public:
    date(void) { };
    friend ifstream& operator >>(ifstream& stream,
        date *date_object);
    friend ostream& operator <<(ostream& stream, date date_object);
private:
    int month;
    int day;
    int year;
};

ifstream& operator >>(ifstream& stream, date *date_object)
{
    stream >> date_object->month >> date_object->day >>
        date_object->year;

    return(stream);
}

ostream& operator <<(ostream& stream, date date_object)
{
    stream << date_object.month << ' ' << date_object.day << ' ' <<
        date_object.year << endl;

    return(stream);
}

void main(void)
{
    ifstream datefile("DATES.DAT");
    date dateinfo;

    while (!datefile.eof())
    {
        datefile >> &dateinfo;
        if (!datefile.eof())
            cout << dateinfo;
    }
}
```

După cum se observă, regulile de suprapunere a operatorilor de inserție și extracție pentru operații I/O bazate pe fișiere nu sunt complicate.

#### ATENȚIE LA FIȘIERELE BINARE

Rețineți că nu trebuie să folosiți operatorii de inserție și extracție pentru a scrie valori binare (de obicei, valori de tip real) pe un fișier. Așa cum cunoașteți deja, operatorul de inserție va converti astfel de valori în caractere ASCII, care pot cauza erori la introducerea valorilor. În schimb, trebuie folosite funcțiile membru *read* și *write* pentru a realiza operații I/O. De exemplu, următorul program,



FLOATVAL.CPP, suprapune operatorul de inserție pentru a scrie câteva structuri ce conțin valori reale în fișierul FLOATVAL.DAT. Pentru a afișa valorile, programul folosește funcția membru *write*:

```
#include <fstream.h>

class values {
public:
    values(float a, float b, float c);
    friend ostream& operator <<(ostream& stream,
        values *value_object);
private:
    float a;
    float b;
    float c;
};

values::values(float a, float b, float c)
{
    values::a = a;
    values::b = b;
    values::c = c;
}

ostream& operator <<(ostream& stream, values *value_object)
{
    stream.write((char *) value_object, sizeof(class values));

    return(stream);
}

void main(void)
{
    ofstream value_file("FLOATVAL.DAT");

    values a(1.1, 2.2, 3.3);
    values b(11.11, 22.22, 33.33);
    values c(111.111, 222.222, 333.333);

    value_file << &a;
    value_file << &b;
    value_file << &c;
}
```

În mod similar, următorul program, FLOATIN.CPP, suprapune operatorul de extracție pentru a citi valori reale din fișierul FLOATVAL.DAT într-o variabilă de tip structură. Programul va continua să citească și să afișeze valorile membrilor structurii până la întâlnirea sfârșitului de fișier:

```
#include <fstream.h>

class values {
```

```
public:
    values(void) {};
    friend ifstream& operator >>(ifstream& stream,
        values *value_object);
    friend ostream& operator <<(ostream& stream,
        values *value_object);
private:
    float a;
    float b;
    float c;
};

ifstream& operator >>(ifstream& stream, values *value_object)
{
    stream.read((char *) value_object, sizeof(class values));

    return(stream);
}

ostream& operator <<(ostream& stream, values *value_object)
{
    stream << value_object->a << ' ' << value_object->b << ' ' <<
        value_object->c << endl;

    return(stream);
}

void main(void)
{
    ifstream value_file("FLOATVAL.DAT");

    values a, b, c;

    value_file >> &a;
    cout << &a;

    value_file >> &b;
    cout << &b;

    value_file >> &c;
    cout << &c;
}
```

La compilarea și execuția acestui program, pe ecran va apărea:

```
C:\> FLOATIN <ENTER>
1.1 2.2 3.3
11.11 22.22 33.33
111.111 222.222 333.333
```

## REZUMAT

Clasele cu operatori I/O intrinseci facilitează înțelegerea programelor. Așa cum ați văzut, crearea și folosirea unor astfel de clase nu este dificilă. Înainte de a trece la următorul capitol, asigurați-vă că ați învățat următoarele:

- ✓ Pentru a crea o clasă cu operatori I/O intrinseci se suprapun operatorii de inserție (<<) și extracție (>>).
- ✓ Suprapunerea operatorilor pentru crearea unei clase cu operatori I/O intrinseci nu diferă de suprapunerea operatorilor de adunare sau scădere; suprapunerea afectează un anumit tip de date.
- ✓ Operatorul suprapus trebuie declarat *friend* în cadrul clasei pentru a putea avea acces la membrii de tip *private* ai clasei.
- ✓ Suprapunerea operatorilor pentru crearea claselor cu operatori I/O intrinseci este similară creării unui manipulator - se definește o funcție ce se execută când operatorul este folosit cu un anumit tip de date.
- ✓ Funcția apelată de operatorul suprapus nu este o funcție membru a clasei, ci este declarată în afara clasei.
- ✓ Valoarea returnată și primul parametru al unei funcții apelate de un operator suprapus trebuie să fie de tipul *stream*.
- ✓ Clasa pentru care se creează operatori I/O intrinseci este definită de al doilea parametru al funcției apelate de operatorul suprapus.
- ✓ La fel ca *cin* și *cout*, se pot folosi și *stream-urifisier* pentru operații I/O intrinseci claselor, prin simpla suprapunere a operatorilor pentru obiecte de tipul *ofstream*.
- ✓ Operatorii de inserție și extracție nu trebuie folosiți cu date binare. În locul lor se folosesc funcțiile membru *read* și *write*.

## CAPITOLUL 17 CREAREA UNEI BIBLIOTECI DE CLASE

În sensul cel mai simplu, o *bibliotecă* este o colecție de funcții folosite de programe pentru a realiza anumite sarcini. Biblioteca standard C/C++, de exemplu, conține funcții ce permit programelor să prelucreze șiruri de caractere, să lucreze cu fișiere, să aloce memorie ș.a.m.d. Pentru a folosi o bibliotecă, trebuie inclus, de obicei, un fișier antet la începutul programului pentru a specifica prototipurile de funcții. Apoi, la editarea programului, se specifică numele bibliotecii, pentru a permite editorului de legături să localizeze funcțiile. În cazul unei biblioteci standard, editorul de legături caută automat funcțiile în anumite fișiere. Folosirea bibliotecilor elimină nevoia de a scrie și a testa un volum considerabil de cod, deoarece funcțiile din bibliotecă sunt testate și funcționează corect.

O *bibliotecă de clase* este o bibliotecă ce conține funcțiile membru ale unor anumite clase. Pe măsură ce construiți definițiile de clase, veți constata și o anumită clasă, necesară unui program, este adesea utilă și altui program. Punând împreună toate funcțiile membru într-o bibliotecă de clase, refolosirea claselor în două sau mai multe programe devine foarte simplă. Mulți proiectanți de software livrează biblioteci de clase. De exemplu, dacă programați în C++ pentru mediul Windows, există biblioteci de clase ce vă permit să realizați operații cu ferestre. Există, de asemenea, biblioteci de clase pentru aplicații multimedia. În fiecare caz, bibliotecile conțin funcții membru și definiții asociate unor clase C++.

Acest capitol examinează etapele ce trebuie efectuate pentru a crea propriile biblioteci de clase. În momentul încheierii acestui capitol, veți cunoaște:

- ♦ Ce este, cum se creează și cum se folosește o bibliotecă de programe obiect
- ♦ Ce este o bibliotecă de clase, cum se creează și cum se folosește.

### CREAREA ȘI FOLOSIREA UNEI BIBLIOTECI DE OBIECTE

Înainte de a examina etapele necesare creării unei biblioteci de clase, este util de a înțelege modul de creare a unei *biblioteci de obiecte*. O bibliotecă de obiecte este un fișier ce conține codul compilat al unui set de funcții. La crearea programelor se pot folosi aceste funcții precompilate pentru a economisi timp,

efort de programare și testare. Să construim acum biblioteca de obiecte STRFUNC.LIB ce conține funcțiile *string\_copy* și *string\_length*. Pentru început, creăm fișierul STRFUNC.CPP ce conține următoarele:

```
void string_copy(char *target, char *source)
{
    while (*target++ = *source++)
        ;
}

int string_length(char *string)
{
    for (int length = 0; *string++; length++)
        ;

    return(length);
}
```

Se compilează apoi acest fișier sursă pentru a obține fișierul obiect STRFUNC.OBJ. Dacă folosiți Borland C++, apelați la următoarea linie de comandă pentru a crea fișierul obiect:

```
C:\> BCC -c STRFUNC.CPP <ENTER>
```

Dacă folosiți Microsoft Visual C++, atunci linia de comandă va fi:

```
C:\> CL -c STRFUNC.CPP <ENTER>
```

Apoi, folosind programe de bibliotecă furnizate de compilator, creați fișierul de bibliotecă STRFUNC.LIB. Pentru Borland C++, folosiți următoarea comandă TLIB pentru a crea fișierul de bibliotecă:

```
C:\> TLIB STRFUNC.LIB +STRFUNC.OBJ <ENTER>
```

Dacă lucrați cu Microsoft Visual C++, folosiți comanda LIB:

```
C:\> LIB STRFUNC.LIB +STRFUNC.OBJ: <ENTER>
```

**Observație:** La folosirea utilitarului Microsoft LIB, nu uitați să scrieți simbolul „;” la sfârșitul liniei de comandă, ceea ce va determina folosirea unor parametri prestabiliți. În caz contrar, se va cere introducerea unor nume de fișiere.

Pentru facilita specificarea prototipurilor de funcții, se poate crea un fișier antet ce conține prototipurile funcțiilor de bibliotecă. Să creăm, deci, fișierul antet STRFUNC.H ce conține următoarele prototipuri:

```
void string_copy(char *target, char *source);

int string_length(char *string);
```

Următorul program, STRDEMO.CPP, folosește funcțiile de bibliotecă *string\_length* și *string\_copy*:

```
#include <iostream.h>
#include "strfunc.h" // string library header

void main(void)
{
    char source[64] = "Success with C++";
    char target[64];
    char chapter[64];

    cout << "The string " << source << " has " <<
        string_length(source) << " characters" << endl;

    string_copy(chapter, "Chapter 17");

    cout << "This is " << chapter << endl;

    string_copy(target, source);

    cout << "I am reading " << target << endl;
}
```

Dacă compilați (sau legați) acest program, trebuie specificat numele bibliotecii STRFUNC.LIB. Pentru compilatorul Borland C++, se poate compila și lega programul folosind următoarea comandă:

```
C:\> BCC STRDEMO.CPP STRFUNC.LIB <ENTER>
```

Pentru compilatorul Microsoft Visual C++, compilarea și legarea se realizează astfel:

```
C:\> CL STRDEMO.CPP STRFUNC.LIB <ENTER>
```

La execuția programului, pe ecran va apărea:

```
C:\> STRDEMO <ENTER>
The string Success with C++ has 16 characters
This is Chapter 17
I am reading Success with C++
```



#### CREAREA UNEI BIBLIOTECI DE OBIECTE

O bibliotecă de obiecte este un fișier ce conține codul compilat al unui set de funcții. La crearea programelor, aceste funcții precompilate pot fi folosite pentru a economisi timp și efort de programare și testare. Pentru a crea o bibliotecă de obiecte proprie, realizați pașii următori:

1. Creați un fișier sursă ce conține funcții asociate.
2. Compilați codul sursă pentru a produce un fișier obiect.

3. Folosiți un program utilitar pentru a crea fișierul bibliotecă.
4. Creați un fișier antet ce conține prototipurile fiecărei funcții de bibliotecă.

Pentru a folosi funcțiile de bibliotecă în programe, urmați pașii de mai jos:

1. Includeți fișierul antet corespunzător la începutul programului.
2. Folosiți în programul sursă una sau mai multe funcții de bibliotecă.
3. Legați programul cu fișierul bibliotecă.

## CREAREA UNEI BIBLIOTECI DE CLASE

Așa cum ați învățat, o bibliotecă de cod obiect conține una sau mai multe funcții ce pot fi legate la program. În mod asemănător, o bibliotecă de clase conține codul obiect al uneia sau mai multor funcții. Într-o bibliotecă de clase, însă, funcțiile corespund unei clase sau unei mulțimi de clase asociate. Așa cum veți vedea, algoritmul ce trebuie urmat pentru a crea o bibliotecă de clase nu diferă de cel discutat anterior pentru crearea unei biblioteci de cod obiect:

1. Plasați declarația clasei și funcțiile membru în fișierul sursă.
2. Compilați fișierul sursă pentru a crea un fișier obiect.
3. Folosiți un program utilitar pentru a crea fișierul bibliotecă.
4. Creați un fișier antet ce conține declarația clasei.

Pentru a folosi în program biblioteca de clase, realizați următoarele:

1. Includeți fișierul antet corespunzător la începutul programului.
2. Utilizați unul sau mai multe obiecte în program.
3. Compilați fișierul sursă al programului și legați-l cu fișierul de bibliotecă al claselor.

După cum cunoașteți, multe programe folosesc interfețe pe bază de meniuri. În acest scop s-ar putea crea o bibliotecă cu o clasă *menu* ce ar putea fi folosită de multe alte programe. Următoarea declarație de exemplu, creează o clasă *menu*:

```
struct option {           // each menu option has option
    char menu_text[64];    // text and a related keystroke
    char keystroke;
};

class menu {
public:
    menu(char *title, option *menu_options, int foreground,
        int background);
    void show_menu(void);
    int get_option(void);
private:
```

## 17: Crearea unei biblioteci de clase

```
char title[64];           // menu title
option options[7];        // array of menu option strings
int number_of_options;    // number of menu options
int foreground;           // menu foreground color
int background;           // menu background color
int row;                  // row to start menu display
int column;               // column to start menu display
void set_colors(int foreground, int background);
void set_row_and_column(int row, int column);
};
```

Clasa *menu* acceptă meniuri de comenzi ce pot conține maxim 7 opțiuni. Pentru a crea un obiect de tip *menu*, programul trebuie să specifice un tablou de opțiuni, un tablou de taste și culorile de meniu, cum se arată în continuare:

```
option menu_a_options[7] = {"Option One", 'A'},
                           {"Option Two", 'B'},
                           {"Option Three", 'C'};

menu a("Very Long Menu Title", menu_a_options, 37, 44);
```

În exemplul dat, culorile folosite vor fi text alb (37) pe fundal albastru (44). Aceste coduri de culoare sunt definite de driverul ANSI.

**Observație:** Programul folosește secvențele escape ANSI pentru a selecta culoarea ecranului și a poziționa cursorul. Dacă lucrați într-un mediu PC, trebuie mai întâi să instalați driverul ANSI.SYS pentru ca sistemul să răspundă corect la secvențele escape de culoare. Mai multe informații asupra folosirii acestor secvențe în programele C++ le puteți găsi în cartea „Jamsa's 1001 C/C++ Tips”, Jamsa Press, 1993.

Funcția constructor a clasei *menu* determină linia și coloana de plasare a opțiunilor de meniu, pentru ca acestea să fie centrate pe ecran în momentul afișării. În acest scop este nevoie de determinarea numărului de opțiuni din meniu și de lungimea textului fiecărei opțiuni.

Pentru a crea biblioteca clasei *menu*, plasați declarația clasei *menu* în fișierul MENU.CPP, cum se arată mai jos:

```
#include <iostream.h>
#include <string.h>
#include <ctype.h>

struct option {
    char menu_text[64];
    char keystroke;
};

class menu {
public:
    menu(char *title, option *menu_options,
```

```

    int foreground, int background);
void show_menu(void);
int get_option(void);
private:
char title[64];           // menu title
option_options[7];       // array of menu option strings
int number_of_options;    // number of menu options
int foreground;           // menu foreground color
int background;          // menu background color
int row;                  // row to start menu display
int column;               // column to start menu display
void set_colors(int foreground, int background);
void set_row_and_column(int row, int column);
};

menu::menu(char *title, option *menu_options,
            int foreground, int background)
{
    strcpy(menu::title, title);

    int number_of_options = 0;
    int longest_option = strlen(title);

    for (int i = 0; i < 7; i++)
    {
        int option_length = strlen(menu_options[i].menu_text);

        if (option_length > longest_option)
            longest_option = option_length;

        strcpy(menu::options[i].menu_text,
            menu_options[i].menu_text);
        menu::options[i].keystroke = menu_options[i].keystroke;
        if (options[i].menu_text)
            number_of_options++;
    }

    menu::number_of_options = number_of_options;
    menu::foreground = foreground;
    menu::background = background;

    column = (79 - longest_option) / 2;

    // Center options plus title, prompt and 2 blank lines
    row = (24 - (4 + number_of_options)) / 2;
}

void menu::set_colors(int foreground, int background)
{
    cout << "\033[" << foreground << ';' << background << 'm' <<
        endl;
}

```

```

    cout << "\033[2J";
}

void menu::set_row_and_column(int row, int column)
{
    cout << "\033[" << row << ';' << column << 'H';
}

void menu::show_menu(void)
{
    set_colors(foreground, background);
    set_row_and_column(row, column);

    cout << title << endl;

    for (int i = 0; i < number_of_options; i++)
    {
        set_row_and_column(row + 2 + i, column);
        cout << options[i].keystroke << ' ' <<
            options[i].menu_text;
    }

    set_row_and_column(row + i, column);
    cout << "Enter choice: ";
}

int menu::get_option(void)
{
    char option;
    int done = 0, choice;

    while (!done)
    {
        show_menu();
        option = cin.get();
        option = toupper(option);

        for (int i = 0; i < number_of_options; i++)
            if (option == options[i].keystroke)
            {
                choice = i;
                done = 1;
            }

        while (cin.get() != '\n')
            ; // Eat characters up to and including \n
    }

    return (choice);
}

```

Compilați apoi fișierul sursă pentru a crea un fișier obiect. Dacă lucrați cu compilatorul Borland C++, folosiți comanda:

```
C:\> BCC -c MENU.CPP <ENTER>
```

În caz că lucrați cu compilatorul Microsoft Visual C++, folosiți următoarea linie de comandă pentru compilarea fișierului:

```
C:\> CL -C MENU.CPP <ENTER>
```

Fișierul de bibliotecă se creează, în funcție de compilator, folosind TLIB sau LIB:

```
C:\> TLIB MENU.LIB +MENU.OBJ <ENTER>
```

```
C:\> LIB MENU.LIB +MENU.OBJ; <ENTER>
```

**Observație:** Nu uitați să includeți simbolul ';' la sfârșitul comenzii LIB pentru a evita obligativitatea specificării numelor de fișiere.

Următorul program, MENUDEMO.CPP, creează și afișează diferite opțiuni de meniu folosind biblioteca clasei *menu*:

```
#include <iostream.h>
#include "menu.h"

void main(void)
{
    int choice;

    option menu_a_options[7] = {"Option One", 'A'},
                                {"Option Two", 'B'},
                                {"Option Three", 'C'};

    menu a("Very Long Menu Title", menu_a_options, 37, 44);

    choice = a.get_option();

    cout << endl << "You selected option " << choice+1 << endl;
    cout << "Press Enter to continue...";
    while (cin.get() != '\n')
        ;

    option menu_b_options[7] = {"Perform Payroll", 'P'},
                                {"Update Accounts Payable", 'A'},
                                {"Print Checks", 'C'},
                                {"Quit", 'Q'};

    menu b("Not So Quicken", menu_b_options, 37, 41);

    choice = b.get_option();

    cout << endl << "You selected option " << choice+1 << endl;
```

```
cout << "Press Enter to continue...";
while (cin.get() != '\n')
    ;

// Restore white on black screen
cout << "\033[37;40m\033[2J";
}
```

Pentru a compila și lega acest program folosind Borland C++, dați următoarea comandă:

```
C:\> BCC -MENUDEMO.CPP MENU.LIB
```

Dacă lucrați cu Microsoft Visual C++, folosiți comanda:

```
C:\> CL MENUDEMO.CPP MENU.LIB
```

### REZUMAT

O bibliotecă este o colecție de funcții precompilate folosite de programe pentru a realiza sarcini specifice. În cazul unei biblioteci de clase, funcțiile corespund unei anumite clase sau unei mulțimi de clase asociate. În acest capitol s-au examinat etapele ce trebuie efectuate pentru a crea o bibliotecă de clase, precum și etapele necesare pentru a folosi o astfel de bibliotecă în programe. În Capitolul 18 veți folosi aceste etape pentru a construi o bibliotecă de clase de liste dinamice. Înainte de a trece la Capitolul 18, asigurați-vă că ați învățat următoarele:

- ✓ O bibliotecă de funcții obiect este un fișier ce conține codul compilat al unui set de funcții.
- ✓ Pentru a crea o bibliotecă obiect se scriu sau se assemblează funcțiile într-un fișier, se compilează fișierul, se folosește programul utilitar ce însoțește compilatorul pentru crearea fișierului de bibliotecă și se construiește un fișier antet ce conține prototipurile funcțiilor.
- ✓ Pentru a folosi funcțiile de bibliotecă în program, se include fișierul antet corespunzător la începutul programului, se folosesc funcțiile de bibliotecă în codul programului sursă și se leagă programul cu fișierul de bibliotecă.
- ✓ Pentru a crea o bibliotecă de clase se plasează declarația clasei și a funcțiilor membru în fișierul sursă, se compilează fișierul sursă pentru a crea fișierul obiect, se folosește un program utilitar pentru a crea fișierul de bibliotecă și se construiește un fișier antet ce conține declarația clasei.
- ✓ Pentru a folosi o bibliotecă de clase se include fișierul antet corespunzător la începutul programului, creează unul sau mai multe obiecte în program, apoi se compilează și se leagă programul sursă la fișierul bibliotecă.

## CAPITOLUL 18

### CREAREA UNEI CLASE DE LISTE DINAMICE

Majoritatea programelor folosesc informații ce pot fi memorate și prelucrate sub formă de liste dinamice. Așa cum ați citit pe scurt în Capitolul 7, listele dinamice sunt de preferat tablourilor deoarece folosesc eficient memoria disponibilă a calculatorului. Dacă ați lucrat în trecut cu liste cu legături simple sau duble, știți deja că astfel de liste sunt formate din *noduri*, ce conțin date, și *pointeri*, ce conectează nodurile în listă. Datorită utilizării pe scară largă a listelor înlănțuite, programatorii încearcă întotdeauna modalități de a crea *liste generice* ce pot fi ușor adaptate pentru a satisface necesitățile diferitelor structuri de date folosite în programul curent sau chiar în alte programe.

Acest capitol vă ajută să creați o astfel de listă generică. Pentru a putea comuta lista de la o aplicație la alta, se creează șabloane de clasă pentru liste dinamice. În acest capitol se folosesc concepte prezentate în capitolele anterioare. Aveți răbdare să exersați cu fiecare din programele prezentate. La început, sintaxa C++ pentru crearea listelor generice pare descurajantă. Totuși, dacă citiți comentariile ce însoțesc fiecare exemplu, veți vedea că prelucrarea acestor liste nu este complicată. În momentul terminării acestui capitol, veți cunoaște:

- ♦ Ce este o listă cu legături simple
- ♦ Cum se folosește un șablon de clasă pentru a accepta variabile de diferite tipuri
- ♦ Ce este o listă cu legături duble
- ♦ Cum se parcurge o listă și cum se contorizează apariția unei anumite valori în listă
- ♦ Care este avantajul folosirii bibliotecilor de clase tip șabloane

#### ANALIZA UNEI LISTE CU LEGĂTURI SIMPLE

O *listă cu legături* este un grup de articole aranjate în secvență și *legate* împreună, astfel încât fiecare membru sau nod să indice spre următorul articol din listă. Această structură este utilă îndeosebi pentru aplicații de tip listă, cum sunt bazele de date.

Înainte de a intra în detaliile unei liste generice, să analizăm un exemplu simplu pentru a ne obișnui cu listele înlanțuite. Pentru început, să creăm o listă cu legături simple ale cărei noduri să memoreze valorile de la 1 la 100.

Pentru a crea lista cu legături, programul folosește o clasă denumită *list* și o structură denumită *node*. Clasa *list* conține variabilele membru *first* și *last* care indică începutul și sfârșitul listei:

```
class list {
public:
    list(void) { first.next = NULL; last = &first; };
    void show_list(void);
    void append_node(node *new_node);
private:
    node first;
    node *last;
};
```

După cum se poate constata, clasa conține funcțiile membru *append\_node* și *show\_list*. Funcția *append\_node* atașează nodul specificat la sfârșitul listei înlanțuite. De asemenea, funcția *show\_list* parcurge lista, afișând valoarea din fiecare nod. Structura *node* conține o valoare de tip *int* și un pointer la următorul nod din listă:

```
struct node {
    int data;
    struct node *next;
};
```

Următorul program, SIMPLE.CPP, implementează lista simplă cu legături folosind clasa *list* și structura *node*:

```
#include <iostream.h>

struct node {
    int value;
    struct node *next;
};

class list {
public:
    list(void) { first.next = NULL; last = &first; };
    void show_list(void);
    void append_node(node *new_node);
private:
    node first;
    node *last;
};

void list::show_list(void)
```

```
{
    node *current_node = first.next;

    while (current_node)
    {
        cout << current_node->value << endl;
        current_node = current_node->next;
    }
}

void list::append_node(node *new_node)
{
    last->next = new_node;
    last = new_node;
    last->next = NULL;
}

void main(void)
{
    list single;
    node *new_node;

    for (int i = 1; i <= 100; i++)
    {
        new_node = new node;
        new_node->value = i;
        single.append_node(new_node);
    }
    single.show_list();
}
```

După cum se poate vedea, programul atașează valorile la sfârșitul listei, folosind funcția *append\_node*. După ce toate valorile au fost atribuite listei, programul afișează conținutul acesteia.



### SĂ ÎNȚELEGEM FUNCȚIA APPEND\_NODE

La prima vedere, funcția *append\_node* (de inserare la sfârșitul listei) poate fi derutantă, dar este foarte simplă. Rețineți că variabila membru *last* a clasei *list* indică spre sfârșitul listei. De asemenea, variabila pointer *next* din structura *node* indică întotdeauna spre la următorul nod din listă.

Când introducem un nou nod în listă, acel nod trebuie să devină ultimul în listă. Aceasta se realizează atribuind noul nod variabilei membru *last*:

```
last = new_node;
```



Dar, așa cum observați, aceasta nu este prima instrucțiune în funcția `append_node`. Reamintiți-vă că noul nod trebuie să devină ultimul nod din listă. Aceasta înseamnă că ultimul nod vechi va deveni penultimul nod în listă. Din acest motiv, vechiul nod ultim trebuie să indice noul nod ca fiind următorul în listă. Pentru aceasta trebuie să facem ca `next` să punteze la `new_node` înainte de a-l face pe nodul `new_node` ultimul (`last`) în listă:

```
last->next = new_node;
```

În fiecare listă cu legături trebuie să cunoaștem când s-a ajuns la ultimul nod în listă. O practică de programare obișnuită este de a folosi valoarea NULL pentru a indica sfârșitul listei. Programul SIMPLE.CPP realizează acest lucru atribuind valoarea NULL pointerului `next` al noului nod, care între timp a devenit ultimul nod (`last`):

```
last->next = NULL;
```

Acum putem înțelege mai ușor funcția constructor `list`:

NULL indică lipsa următorului nod  
Primul și ultimul nod coincid

```
list(void) { first.next = NULL; last = &first; };
```

Când se inițializează lista, nodul `last` este același cu nodul `first`. De asemenea, pointerul `next` este NULL deoarece nu există următorul nod. Primul nod este primul, ultimul și singurul nod în listă când aceasta este inițializată de funcția constructor.

Pentru a atribui valori listei cu legături, programul anterior a folosit următoarele instrucțiuni pentru alocarea unui nod și atribuirea unei valori componentei nodului:

```
for (int i = 1; i <= 100; i++)
{
    new_node = new node;
    new_node->value = i;
    single.append_node(new_node);
}
```

Dacă se schimbă tipul `node` din structură în clasă, se poate reduce numărul de instrucțiuni necesare creării listei la următoarele:

```
for (int i = 1; i <= 100; i++)
    single.append_node(new node(i));
```

În acest caz, programul atașează obiectul pointer inițializat la sfârșitul listei cu legături într-un singur pas. Următorul program, NODECLAS.CPP, folosește clasa `node` pentru a construi lista cu legături:

```
#include <iostream.h>
```

```
class node {
```

```
public:
    node(int value) { node::value = value; };
    node(void) { };
    void show_value(void) { cout << value << endl; };
    struct node *next;
private:
    int value;
};
```

```
class list {
public:
    list(void) { first.next = NULL; last = &first; };
    void show_list(void);
    void append_node(node *new_node);
private:
    node first;
    node *last;
};
```

```
void list::show_list(void)
{
    node *current_node = first.next;

    while (current_node)
    {
        current_node->show_value();
        current_node = current_node->next;
    }
}
```

```
void list::append_node(node *new_node)
{
    last->next = new_node;
    last = new_node;
    last->next = NULL;
}
```

```
void main(void)
{
    list single;

    for (int i = 1; i <= 100; i++)
        single.append_node(new node(i));

    single.show_list();
}
```

După cum se observă, clasa `node` furnizează funcții constructor și o funcție de afișare a valorilor din noduri.

## FOLOSIREA UNUI ȘABLON DE CLASĂ

În programele precedente, lista cu legături conținea numai valori de tip *int*. Folosind următorul șablon de clasă structura *node* poate fi modificată pentru a accepta valori de tip *int*, *float*, *long* ș.a.m.d.:

```
template<class T> class node {
public:
    node(T value) { node::value = value; };
    node(void) { };
    void show_value(void) { cout << value << endl; };
    struct node<T> *next;
private:
    T value;
};
```

Următorul program, NEW\_LIST.CPP, folosește șablonul de clasă *node* pentru a crea 3 liste, una pentru valori de tip *int*, una pentru valori de tip *float* și alta pentru valori de tip *char*:

```
#include <iostream.h>

template<class T> class node {
public:
    node(T value) { node::value = value; };
    node(void) { };
    void show_value(void) { cout << value << endl; };
    struct node<T> *next;
private:
    T value;
};

template<class T> class list {
public:
    list(void) { first.next = NULL; last = &first; };
    void show_list(void);
    void append_node(node<T> *new_node);
private:
    node<T> first;
    node<T> *last;
};

template<class T> void list<T>::show_list(void)
{
    node<T> *current_node = first.next;
    while (current_node)
    {
        current_node->show_value();
        current_node = current_node->next;
    }
}
```

```

    }
}

template<class T> void list<T>::append_node(node<T> *new_node)
{
    last->next = new_node;
    last = new_node;
    last->next = NULL;
}

void main(void)
{
    list<int> single;
    list<float> values;
    list<char> letters;

    for (int i = 1; i <= 10; i++)
        single.append_node(new node<int> (i));

    single.show_list();

    for (i = 0; i < 10; i++)
        values.append_node(new node<float> (0.1 * i));

    values.show_list();

    for (i = 'A'; i <= 'Z'; i++)
        letters.append_node(new node<char> (i));

    letters.show_list();
}
```

Se poate observa că, folosind șablonul de clasă *node*, programul poate crea ușor liste cu legături capabile să memoreze date întregi, reale sau caractere.

**Observație:** Dacă aveți probleme în înțelegerea șabloanelor de clasă din programul NEW\_LIST.CPP, recitiți Capitolul 6. Singura diferență reală între programele NEW\_LIST.CPP și NODECLAS.CPP rezidă în funcția main. Celelalte componente ale celor două programe sunt identice, excepție făcând faptul că în NODECLAS.CPP clasele și funcțiile sunt implementate prin șabloane. Încercați să analizați programul NEW\_LIST.CPP fără a lua în considerare nici un aspect legat de șabloane.

## CÂND PROGRAMELE ACCEPTĂ LISTE CU LEGĂTURI DUBLE

Pentru inserția și ștergerea elementelor dintr-o listă cu legături, multe operații se pot simplifica folosind o listă cu legături duble. În sensul cel mai simplu, o listă cu legături duble conține date și doi pointeri, unul care indică nodul următor și altul nodul precedent.

```
template<class T> class node {
public:
    node(T value) { node::value = value; };
    node(void) { };
    void show_value(void) { cout << value << endl; };
    struct node<T> *next;
    struct node<T> *previous;
private:
    T value;
};
```

Să observăm că singura diferență între acest șablon și șablonul *node* din *NEW\_LIST.CPP* este apariția pointerului *previous*:

```
struct node<T> *previous;
```

Următorul program, *DOUBLY.CPP*, modifică structura *node* și funcțiile membru ale clasei *list* pentru a accepta o listă cu legături duble:

```
#include <iostream.h>
```

```
template<class T> class node {
public:
    node(T value) { node::value = value; };
    node(void) { };
    void show_value(void) { cout << value << endl; };
    struct node<T> *next;
    struct node<T> *previous;
private:
    T value;
};
```

```
template<class T> class list {
public:
    list(void) { first.next = NULL; last = &first;
                first.previous = NULL; };
    void show_list(void);
    void append_node(node<T> *new_node);
    void show_reverse(void);
private:
    node<T> first;
    node<T> *last;
};
```

```
template<class T> void list<T>::show_list(void)
{
    node<T> *current_node = first.next;

    while (current_node)
    {
        current_node->show_value();
```

```
        current_node = current_node->next;
    }
}

template<class T> void list<T>::show_reverse(void)
{
    node<T> *current_node = last;

    while (current_node->previous)
    {
        current_node->show_value();
        current_node = current_node->previous;
    }
}
```

```
template<class T> void list<T>::append_node(node<T> *new_node)
{
    last->next = new_node;
    new_node->previous = last;
    last = new_node;
    last->next = NULL;
}
```

```
void main(void)
{
    list<int> single;
    list<float> values;
    list<char> letters;

    for (int i = 1; i <= 10; i++)
        single.append_node(new node<int> (i));

    single.show_list();

    for (i = 0; i < 10; i++)
        values.append_node(new node<float> (0.1 * i));

    values.show_list();

    for (i = 'A'; i <= 'Z'; i++)
        letters.append_node(new node<char> (i));

    letters.show_list();

    letters.show_reverse();
}
```

După cum se observă, clasa adaugă funcția *show\_reverse* care afișează conținutul listei în ordine inversă. Deoarece fiecare nod conține acum un pointer la nodul precedent, parcurgerea listei în ordine inversă este foarte simplă.

## Succes cu C++

Funcția `append_node` este similară celei din programele anterioare, cu excepția faptului că apare pointerul `previous` ce indică nodul precedent din listă.

De asemenea, funcția constructor `list` trebuie să inițializeze atât `previous` cât și `next` la valoarea `NULL`. Ca și în cazul unei liste cu legături simple când este inițializată de funcția constructor `list`, primul nod al unei liste cu legături duble este, în momentul inițializării, primul, ultimul și singurul nod în listă - nu există noduri următoare (`next`) sau precedente (`previous`).

Fiecare din programele precedente au adăugat elemente la lista cu legături prin simpla atașare a acestora la finalul listei. Următorul program, `INSERT.CPP`, include în clasa `list` și funcția membru `insert_node` ce permite inserția valorilor în listă în ordine crescătoare. Folosind o listă cu legături duble, programul poate insera cu ușurință un element în listă:

```
#include <iostream.h>
#include <stdlib.h>

template<class T> class node {
public:
    node(T value) { node::value = value; };
    node(void) { };
    void show_value(void) { cout << value << endl; };
    T get_value(void) { return(value); };
    struct node<T> *next;
    struct node<T> *previous;
private:
    T value;
};

template<class T> class list {
public:
    list(void) { first.next = NULL; last = &first;
                first.previous = NULL; };
    void show_list(void);
    void append_node(node<T> *new_node);
    void show_reverse(void);
    void insert_node(node<T> *new_node);
private:
    node<T> first;
    node<T> *last;
};

template<class T> void list<T>::show_list(void)
{
    node<T> *current_node = first.next;

    while (current_node)
    {
        current_node->show_value();
```

```
        current_node = current_node->next;
    }
}

template<class T> void list<T>::show_reverse(void)
{
    node<T> *current_node = last;

    while (current_node->previous)
    {
        current_node->show_value();
        current_node = current_node->previous;
    }
}

template<class T> void list<T>::append_node(node<T> *new_node)
{
    last->next = new_node;
    new_node->previous = last;
    last = new_node;
    last->next = NULL;
}

template<class T> void list<T>::insert_node(node<T> *new_node)
{
    node<T> *current_node = first.next;
    node<T> *previous_node = &first;

    while ((current_node) &&
           (current_node->get_value() < new_node->get_value()))
    {
        current_node = current_node->next;
        previous_node = previous_node->next;
    }

    previous_node->next = new_node;
    new_node->previous = previous_node;
    new_node->next = current_node;
}

void main(void)
{
    list<float> values;

    for (int i = 0; i < 10; i++)
        values.insert_node(new node<float> (rand()));

    values.show_list();
}
```

Acțiunea funcției *insert\_node* este evidentă. Instrucțiunea *while* determină parcurgerea listei nod cu nod și compară cu valoarea din *new\_node*:

```
while ((current_node) &&
      (current_node->get_value() < new_node->get_value()))
{
    current_node = current_node->next;
    previous_node = previous_node->next;
}
```

Când valoarea din *current\_node* este mai mică decât valoarea din *new\_node*, ciclul *while* se încheie. Noul nod este inserat cu următoarele instrucțiuni:

```
previous_node->next = new_node;
new_node->previous = previous_node;
new_node->next = current_node;
```

Dacă toate valorile din nod sunt mai mici decât valoarea din *new\_node* atunci funcția *insert\_node* atașează noul nod în finalul listei.

### ADĂUGAREA UNOR MEMBRI AI CLASEI

Programele anterioare au realizat operații simple cu liste, ca inserția sau atașarea unui nod sau afișarea conținutului nodurilor. În multe situații este necesară căutarea unei anumite valori într-o listă cu legături sau efectuarea unei anumite operații cu valorile din noduri. Următorul program, *SEARCH.CPP*, de exemplu, folosește funcția membru *search* pentru a căuta într-o listă cu legături prima apariție a unei valori specificate. În plus, programul folosește funcția membru *cout* pentru a contoriza numărul de apariții ale valorii specificate:

```
#include <iostream.h>
#include <stdlib.h>

template<class T> class node {
public:
    node(T value) { node::value = value; };
    node(void) { };
    void show_value(void) { cout << value << endl; };
    T get_value(void) { return(value); };
    struct node<T> *next;
    struct node<T> *previous;
private:
    T value;
};

template<class T> class list {
public:
    list(void) { first.next = NULL; last = &first;
               first.previous = NULL; };
```

```
void show_list(void);
void append_node(node<T> *new_node);
void show_reverse(void);
void insert_node(node<T> *new_node);
node<T> *search(T value);
int count(T value);
private:
    node<T> first;
    node<T> *last;
};

template<class T> void list<T>::show_list(void)
{
    node<T> *current_node = first.next;

    while (current_node)
    {
        current_node->show_value();
        current_node = current_node->next;
    }
}

template<class T> void list<T>::show_reverse(void)
{
    node<T> *current_node = last;

    while (current_node->previous)
    {
        current_node->show_value();
        current_node = current_node->previous;
    }
}

template<class T> void list<T>::append_node(node<T> *new_node)
{
    last->next = new_node;
    new_node->previous = last;
    last = new_node;
    last->next = NULL;
}

template<class T> void list<T>::insert_node(node<T> *new_node)
{
    node<T> *current_node = first.next;
    node<T> *previous_node = &first;

    while ((current_node) &&
          (current_node->get_value() < new_node->get_value()))
    {
        current_node = current_node->next;
```

```

        previous_node = previous_node->next;
    }

    previous_node->next = new_node;
    new_node->previous = previous_node;
    new_node->next = current_node;
}

template<class T> node<T> *list<T>::search(T value)
{
    node<T> *current_node = first.next;

    while (current_node)
        if (current_node->get_value() == value)
            return(current_node);
        else
            current_node = current_node->next;

    return(NULL);
}

template<class T> int list<T>::count(T value)
{
    node<T> *current_node = first.next;

    int count = 0;

    while (current_node)
    {
        if (current_node->get_value() == value)
            count++;
        current_node = current_node->next;
    }
    return(count);
}

void main(void)
{
    list<float> values;

    for (int i = 0; i < 10; i++)
        values.insert_node(new node<float> (2.5 * i));

    values.show_list();

    if (values.search(7.5))
        cout << "Found the value 7.5" << endl;
    else
        cout << "The value 7.5 was not found" << endl;
}

```

```

    cout << "7.5 was found " << values.count(7.5) << " times" <<
        endl;
}

```

După cum se observă, programul folosește funcția *search* pentru a căuta în listă valoarea 7.5. De asemenea, programul folosește funcția *count* pentru a număra aparițiile valorii 7.5 în listă.

Într-o manieră similară funcției *insert\_node* din programul precedent, funcția *search* parcurge lista astfel:

```

while (current_node)
    if (current_node->get_value() == value)
        return(current_node);
    else
        current_node = current_node->next;

return(NULL);

```

Acum însă, în loc de a căuta o valoare mai mică decât parametrul funcției, cum se proceda în *insert\_node*, se caută o valoare echivalentă *current\_node*. Dacă este găsită o valoare echivalentă, funcția *search* va returna imediat valoarea nodului:

```

if (current_node->get_value() == value)
    return(current_node);

```

Dacă valoarea nu este găsită, funcția *search* returnează NULL.

Funcția *count* este aproape identică cu *search*. Dar, în loc de a reveni în program imediat ce este găsită valoarea, funcția *count* incrementează o variabilă contor de tip *int*:

```

while (current_node)
{
    if (current_node->get_value() == value)
        count++;
    current_node = current_node->next;
}
return(count);

```

Incrementarea variabilei contor

Spre deosebire de celelalte funcții membru examinate, funcția *count* traversează întotdeauna întreaga listă înainte de a reveni în program. Valoarea variabilei contor, care poate fi și zero, este întotdeauna returnată de *count*.

## ȘABLOANE ȘI BIBLIOTECI DE CLASE

În Capitolul 17 ați învățat cum să creați o bibliotecă de clase. Programele prezentate în acest capitol au folosit șabloane pentru a crea liste generice. Așa cum ați învățat în Capitolul 6, în timpul compilării, compilatorul C++ creează clasele

și funcțiile de clasă corecte, pe baza tipurilor specificate la declararea obiectelor. Din păcate, deoarece șabloanele sunt folosite de compilator, este dificil de a crea biblioteci de clase bazate pe șabloane. În cazul programelor anterioare, ar fi mai comodă și mai flexibilă plasarea șabloanelor și funcțiilor într-un fișier antet, decât încercarea de a construi o bibliotecă de clase.

## REZUMAT

Acest capitol a examinat posibilitățile de folosire a șabloanelor pentru a crea o clasă de liste generice cu legături. Programele prezentate în acest capitol ilustrează multe din conceptele discutate în această carte. Este bine să exersați cu fiecare din aceste programe. Înainte de a vă continua aventura cu C++, asigurați-vă că ați învățat următoarele:

- ✓ Într-o listă cu legături simple, fiecare nod indică spre nodul următor, iar ultimul nod indică NULL.
- ✓ Instrucțiunea *list <type>* într-un șablon de clasă determină acceptarea unor variabile de diferite tipuri, indicate de *type*.
- ✓ Într-o listă cu legături duble, fiecare nod are un pointer la următorul nod și un pointer la nodul precedent.
- ✓ Într-un șablon de clasă se pot introduce funcțiile membru *search* și *count* pentru a căuta și număra aparițiile unei anumite valori.
- ✓ Este foarte dificil de a crea biblioteci de șabloane de clase. Introducerea șabloanelor și a funcțiilor într-un fișier antet este mai ușoară și mai flexibilă decât încercarea de a crea o bibliotecă de șabloane de clase.

## Index

### B

backslash (caracterul \), 7  
 backspace (caracterul \b), 4  
 bad (funcție membru), 36-41, 91  
 bell (caracterul \a), 4  
 bibliotecă, *vezi și biblioteci de clase*, 391  
 de obiecte, 391-394  
 binare (fișiere), operații cu, 91-98  
 clase cu operatori I/O intrinseci, 387-390  
 buffer, ieșire prin ~, 41-43  
 golire bufer, 13, 41-43

### C

camuflaj (al informației), 55  
 camuflajul informației, 55  
 caractere  
 caractere de umplere, stabilirea ~, 11-12  
 caractere speciale, 6-8  
 folosire cu I/O pe fișier, 79  
 citire unul câte unul, 29-33  
 numărare, 19-20  
 speciale, 7-9  
 catch (cuvânt-cheie), 336  
 cerr (flux), 6-7  
 cin (stream), 1-5  
 funcții membru, 26-37  
 clasă (constructor de), 55-68, 76-78  
 argumente prestabilite, 65-66  
 cu moștenire, 106-122  
 deschidere fișiere cu ~, 75  
 utilizarea mai multor ~, 64, 65

\ " (ghilimele), 7  
 \' (apostrof), 7  
 \ (backslash), 7  
 \? (semn de întrebare), 7  
 \0 (caracterul nul), 7  
 \000 (valoare în octal), 7  
 \a (alertă sau clopoțel), 4  
 \b (backspace), 4  
 \f (formfeed - avans pagină), 4  
 \n (newline), 4 *vezi și endl*  
 \r (retur de car fără avans hârtie), 7  
 \t (tabulator orizontal), 7  
 \v (tabulator vertical), 7  
 \xhhh (valoare hexazecimală), 7  
 ~ (tilda), 59  
 + (semnul plus), forțarea afișării ~, 18  
 << (operatorul de inserție), 2  
 >> (operatorul de extracție), 2

### A

abstractizare, 228  
 acces aleator (operații cu fișiere), 92-93  
 adresă rezolvată, 267  
 alertă (caracterul \a), 4  
 alias (nume suplimentar), 261-277  
 alinierea ieșirii (stânga și dreapta), 16  
 apostrof (caracterul \'), 7  
 ascunse, obiecte ~, 276

- clasă abstractă, 229
- clasă, bibliotecă de ~, 391-403
  - creare, 394-403
  - definire, 391
- clasă, definiții de ~, 40
- clasă, destructor de ~, 55-68
- clasă, domeniul de vizibilitate al ~, 234-245
- clasă, friend, 153-155, 239-243
- clasă, membri ai ~, 71-75
  - accesul la ~, 54-55
  - de tip *clasic*, 247-249
  - de tip *private*, 51-55, 134-141, 234-239
  - de tip *protected*, 142-145, 258-259
  - de tip *public*, 134-141, 234-239
  - inițializare, 66-68
- clasă, șabloane de ~, 168-177
  - exemple de ~, 406-407
- clase formate (stream I/O), 295-297
- clase, 35-76
  - ce folosesc variabile pointer, 194-195
  - comparate cu
    - obiecte, 41-43
    - structuri, 45
    - uniuni, 45
  - cu I/O neformatat, 295-297
  - cu operatori I/O intrinseci, 377-390
  - cu stream-uri șir, 303-304, 320-321
  - exemple de ~, 45-49
  - istream, 303-304
  - ostream, 303-304
  - pentru flux I/O, 281-284
    - pentru flux șir, 320-321
    - relații între, 301-302
  - pentru I/O formatat, 295-297
  - relații între
    - filebuf, 295-297
    - ifstream, 281-284
    - ios, 281-284
    - iostream, 281-284
    - istream, 281-284
    - ofstream, 281-284
    - ostream, 281-284
    - stdiobuf, 295-297
    - streambuf, 295-297
    - stream-uri șir, 320-321
- istream, 281-284
- ostream, 281-284
- stdiobuf, 295-297
- streambuf, 295-297
- stream-uri șir, 320-321
- stringstream, 303-304
- tablouri de ~, 73-75
- clog, 6
- colectare resturi, 202-203, 371-373
- comparație cu adrese pointer, 267
  - comparație cu variabile pointer, 195-196
  - comparație cu variabile, 264
  - folosire cu structuri, 270-275
  - reguli pentru lucrul cu ~, 265-266
  - sintaxa, 265
- constructor, 60-68, 76-77
  - argumente prestabilite, 65-66
  - moștenire, utilizare cu ~, 106-122
  - pentru deschidere fișiere, 75
  - utilizarea mai multor, 64-65
- controlul lățimii unei valori la ieșirea ~, 10-11
- cout, 1-5
  - funcții membru, 37-38
- cuvinte-cheie
  - catch, 336
  - extern, 245-246
  - operator, 146-148
  - static, 247-249
  - template, 160
  - throw, 336
  - try, 336

## D

- dec (manipulator), 9-12
- declarații, 227-229
  - localizarea variabilelor în, 230-231

- definire, 145
- definiții, 227-229
- delete, operator ~, 184-187
  - suprapunere, 201-207
- destructor, 60-68
  - folosire cu tilda (~), 59
- destructori
  - virtuali, 330-333
- dinamică, legare ~, 327
- domeniu, 227-260
  - al clasei, 234-245
  - al fișierului, 233-234
  - al funcției, 232-233
  - calificatori de ~, 245-250
  - definiție, 228
  - durata de viață a variabilelor, 252-254
    - local, 229-232
- dreapta, aliniere a ieșirii la ~, 16
- durata de viață a variabilelor, 252-254

## E

- ecran (I/O), 1-33
- end of file (sfârșit de fișier), testarea ~, 33
- endl, *vezi și caracterul newline*, 4
- eof (funcție membru), 33, 85, 88, 91-98
- excepție, rutină de testare a ~, 335
- excepții
  - bazate pe clase, 339-342
  - denumirea ~, 337-339
  - incluse ~, 352
  - interfață cu ~, 352
  - mai multe într-o instrucțiune, 348-349
  - nepreluare, 354-356
  - preluare, 336-337
  - prelucrarea ~, 349-351
- returnarea de informații cu ~, 343-348
- rezolvarea (și continuarea) ~, 352-354
- rutină prestabilită, 341-342, 354-356
- tratare ~, 335-357
- extern (cuvânt-cheie), 245-246
- externă, legare ~, 255-256
- extracție, operator de (>>), 2
  - cu fișiere binare, 89-92
  - suprapunere, 150-153

## F

- fail (funcție membru), 36-41, 85, 86-89, 97-98
- filebuf (clasă), 297-302
  - funcții membru, 299
  - relația între clase, 295-297
- fișier, domeniu de vizibilitate al ~, 233-234
- fișier, operații I/O pe ~, 73-99
  - acces aleator, 92-97
  - caractere speciale, 79
  - clase cu operatori I/O intrinseci, 385-390
    - deschidere fișiere, 76-77, 80-81, 82-83, 94-97
    - fișiere binare, 91-98
    - folosire constructor, 76-77
    - ieșire formatată, 77-78
    - ieșire la imprimantă, 97-98
    - moduri de deschidere, 94-97
    - operații de ieșire, 77-84
    - operații de intrare, 84-89
    - operații folosind buffer-ul, 299-302
    - operații read și write, 92
    - rezultatele testării, 84-86
    - specificatori de mod, 94-97
- fișiere, legarea ~, 254-260
- flags, funcție membru, 42-44
- flush (manipulator), 13



## Index

formfeed (caracterul \f), 4  
fragmentare memorie, 201-204  
fstream, 74  
    relații între clase, 281-284  
funcție, domeniu de vizibilitate, 232-233  
funcție, suprapunerea ~, 133-144  
funcții  
    funcții membru în afara definiției, 50, 249-250  
        ce folosesc pointeri ca valori  
        rezultat, 197-198  
        ce folosesc pointeri drept  
        parametri, 192-194  
        în afara definiției de clasă, 50  
        inline, 47-49  
        obiecte ce folosesc ~, 65-66  
        parametri prestabiliți, 141-144  
        referințe ca parametri la ~, 267-270  
        șabloane de ~, 159-178  
        virtuale, 209-226, 323-333  
            destructori, 330-333  
            legări dinamice, 327  
            pure, 224-226  
            tabela de ~, 328-330  
G  
gcount (funcție membru), 19-20  
get (funcție membru), 22-25  
getline (funcție membru), 20-21, 83,  
137-139, 142-143  
ghilimele ("), 7  
global, domeniu ~, 228  
globală, operator de rezoluție ~, 250-252  
golire buffer  
    ieșire prin buffer, 13  
    ieșire stream, 41-43  
good (funcție membru), 36-41, 85-89, 94-97

## H

hex (manipulator), 9-12  
hexazecimal  
    afișare valori în ~, 9-12  
    litere mari, afișare ieșire în ~, 15-16  
    valoare (ca \x1B), 7

## I

I/O (stream-uri), 1-8  
    cerr, 6  
    cin, 1-5  
    clase de, 281-284, 295-294  
    clase formate, 295-297  
        relații între ~, 301  
        fluxuri șir, 320  
    clase neformate, 295-297  
    clog, 6  
    controlul indicatorilor, 14-19  
    cout, 1-5  
    definite, 2  
    prin buffer, 297-302  
    redirectare, 4-5  
    testarea rezultatelor operațiilor,  
    36-41  
    tie (funcție membru), 293-295

I/O prin buffer 297-302  
    cu fișiere 299-302  
ieșire de fișier formatată, 77-78  
ieșire, funcții membru de ~, 27-28  
ieșire, operații de ~  
    fișiere I/O, 74-84  
    stream-uri șir, 312-318, 320  
ifstream, 74  
    relații între clase, 281-284  
ignore, funcție membru ~, 13-14  
imprimantă, ieșire la ~, 97-98  
in\_avail (funcție membru), 299  
indicatori (de stream I/O)

citire și salvare, 31-33  
de control, 14-19  
folosire, 19  
ios:: fixed, 16-17  
ios:: left, 16-17  
ios:: right, 16-17  
ios:: scientific, 16-17  
ios:: showpoint, 16-18  
ios:: showpos, 17-18  
ios:: uppercase, 15-16  
valori prestabilite, refacerea ~,  
18-19

inginerie a programelor, 41-44  
inginerie de programe, 40-41  
inițializare variabile externe, 246-247  
inline, funcții membru ~, 47-49, 249-250  
inserție, operator de ~ (<<), 2  
    folosire cu fișier binar, 89-92  
    suprapunere, 150-153  
interne, legături ~, 255  
intrare (funcții membru de), 25-26  
intrare, operații de ~  
    clase cu operatori I/O intrinseci,  
    384-385  
    fișiere I/O, 84-89  
    stream-uri șir, 304-312, 318-320  
întrebare, semn de (?), 7  
ios, 4

    app, 80, 94-97  
    ate, 80  
    beg, 93  
    binary, 80, 88  
    cur, 93  
    end, 93  
    fixed, 16-17  
    in, 80  
    left, 16  
    nocreate, 80  
    noreplace, 80, 95-96  
    out, 80  
    relații între clase, 281-284

## Index

    cu stream-uri șir, 320-321  
    right, 16  
    scientific, 16-17  
    showbase, 16  
    showpoint, 17-18  
    showpos, 18  
    tie (funcție membru), 293-294  
    trunc, 80  
    uppercase, 16  
iostream, relații între clase, 281-284  
istream, 4  
    istream\_withassign (clasă), 282-283  
    relații între clase, 281-284  
        cu fluxuri șir, 320-321  
istrstream, 304  
    relații între clase, 320-321

## L

legarea  
legarea mai multor fișiere, 254-259  
    dinamică, 327  
    externă, 255  
    internă, 255  
linii de text, citirea de ~, 20-21  
liste cu legături, 401-416  
    cu legături duble, 407-412  
    cu legături simple, 401-405  
    definiția, 401  
local, domeniu ~, 229-232  
    definiție, 227

## M

manipulatori, 8-19, 76-78  
    bază de conversie, stabilirea ~,  
    9-12  
    crearea propriilor ~, 284-293  
    dec, 9-12

flush, 13  
 hex, 9-12  
 oct, 9-12  
 parametri, ce acceptă mai mulți  
 ~, 289-293  
 resetiosflags, 17  
 setfill, 11-12  
 setiosflags, 17  
 setprecision, 12-13  
 setw, 10-11

membru, funcții, 36  
 conflict de nume, rezolvare, 50-51  
 conflict de parametri, rezolvare, 50-51  
 de ieșire, 27-28, 76-78  
 I/O, stream, 36-41, 43-45  
 în afara definiției, 249-250  
 inline, 249-250  
 input, 25-26  
 partajare cod, 250  
 stream-uri șir  
 de ieșire, 316-317  
 de intrare, 311-312

membru, variabile ~, 28

memorie, alocare de ~, 179-207, 298-302  
 colectare de resturi, 202-203  
 dinamică, 184-187  
 stream-uri șir, 315-318  
 efectul ~ asupra zonei de

memorie alocabilă, 359-366  
 problema fragmentării, 202-203  
 set\_new\_handler, 199-201  
 situații de memorie insuficientă, 198-201

moștenire pe mai multe nivele  
 funcții virtuale, 221-226

moștenire, 40, 101-132  
 constructor cu ~, 107-125  
 exemplu de ~, 102-105  
 multiplă, 122-125  
 pe mai multe niveluri, 122-128  
 funcții virtuale, 221-226

## N

neformatate, clase (stream I/O), 295-297

new (operator), 184-187  
 folosire cu stream-uri șir, 315-317  
 inițializare valori, 189-190  
 suprapunere, 201-207

newline (caracterul \n), *vezi și endl*, 4

null (caracterul \0), 7

numărare caractere, 19-20

## O

obiecte, 35-71  
 ascunse, 276  
 atribuire valori (către alte  
 obiecte), 63-65  
 comparate cu clase, 42-43  
 exemple de ~, 43-45  
 folosite cu funcții, 65-66  
 oct (manipulator), 9-12  
 referințe cu ~, 274-275

obiecte, bibliotecă de ~, 391-394

octal  
 afișare valori în ~, 9-12  
 valoare (ex. \033), 7

ofstream, 74  
 folosire constructor, 74-75  
 relații între clase, 281-284

open (funcție membru), 75

operator (cuvânt-cheie), 146-148

operator de rezoluție globală, 250-252

operator, suprapunerea ~, 144-157

operatori I/O intrinseci, clase cu ~, 377-390  
 creare, 381-385  
 fișiere binare, 387-389  
 intrări/ieșiri pe fișiere, 385-389  
 operații de intrare, 384-385

operații I/O, 377-381

orientată obiect, programare ~, 35-41

orizontală (caracter de tabulare \t), 7

ostream, 4  
 ostream\_withassign (clasă),  
 282-283  
 relații între clase, 281-284  
 cu stream-uri șir, 320

ostrstream, 304  
 relații între clase, 320

out\_waiting (funcție membru), 299

## P

parametri prestabiliți de funcții, 141-144

parametri, prestabiliți ai funcției, 141-144

partajat, cod ~, 250

partajate, variabile membru ~, 249

peek (funcție membru), 23-24

plus (semn), forțarea afișării ~, 18

pointer, variabile ~, 187-189  
 aritmetica, 190-191  
 tablouri, tratate ca ~,  
 191-192

clase ~, 194-195

parametri de funcții folosiți ca ~,  
 192-194

probleme cu ~, 196-197

referințe, comparate cu ~, 195-196

structuri, folosite cu ~, 194-195

valoarea de rezultat a funcției,  
 folosită ca ~, 197-198

variabile locale, folosite ca ~,  
 196-197

polimorfism, 209-226, 323-326  
 definiție, 209  
 reguli pentru ~, 220-221

pozitive (numere), afișarea semnului plus, 18

private, membri de clasă tip ~, 51-55, 134-141, 234-239

programare  
 orientată obiect, 35-40  
 scopuri, inginerie de programe,  
 40-41

protected, membri de clasă tip ~, 134-141, 234-239

public, membri de clasă tip ~, 134-141, 234-239

punct zecimal, forțarea afișării în ~, 17-18

pure, funcții virtuale ~, 224-226

put (funcție membru), 74

putback (funcție membru), 24-25

## R

rdstate (funcție membru), 36-41

read (funcție membru), 86-98

referințe, 261-277  
 creare, 263  
 declarare, 261-262  
 descriere, 261-266  
 folosire cu obiecte, 270-275  
 ascunse, 276

retur de car (caracterul \r), 7

rezolvate, adrese, 267

## S

sbumpc (funcție membru), 299

secvențial (operații cu fișiere), 92

seekg (funcție membru), 92-93

seekp (funcție membru), 92-93

set\_new\_handler, 199-200

## Index

setbase (manipulator), 9-12  
setf (funcție membru), 23-33  
setfill (manipulator), 11-12  
setiosflags (manipulator), 9-12, 14-19  
setprecision (manipulator), 12-13  
setw (manipulator), 10-11  
sgetc (funcție membru), 299  
sgetn (funcție membru), 299  
situații de memorie insuficientă, 198-201  
  set\_new\_handler, 199-201  
snextc (funcție membru), 299  
spațiu alb (ignorarea la intrare), 13-14  
spațiu disponibil, 198 *vezi și memorie alocabilă*  
  friend, 153-155, 239-243  
specificatori de memorie, 245  
specificatori de mod (la deschidere fișiere), 80-81, 82-83, 94-97  
sputbackc (funcție membru), 299  
sputc (funcție membru), 299  
sputn (funcție membru), 299  
stânga, aliniere a ieșirii la ~, 16  
static (cuvânt-cheie), 247-249  
static, membri ai clasei de tip ~, 247-249  
stdiobuf, relații între clase, 296-298  
științific, format (afișarea ieșirii în ~), 16-17  
stossc (funcție membru), 299  
str (funcție membru), 318  
streambuf (clasă), 297-302  
  relația de clase, 295-297  
stream-uri I/O. *Vezi I/O (stream-uri)*  
stream-uri șir, 303-321  
  clase de ~, 303-304, 320  
  operații de ieșire, 312-318  
  funcții membru, 316-317  
  operații de intrare, 304-312

  citire valori multiple, 307-311  
  funcții membru, 311-312  
  operații de intrare/ieșire, 318-320  
strstream, 304  
  relații de clase, 320  
structuri  
  clase comparate cu ~, 45  
  folosire referințe cu ~, 270-276  
  variabile pointer ce folosesc ~, 194-195  
suprapunere, 133-158  
  funcții, 134-144  
  operatori I/O, 150-153  
  operatori, 144-157  
  operatorul delete, 201-207  
  operatorul new, 201-207

## Ș

șabloane, 159-178  
  biblioteci de ~, 415-416  
  de clasă, 168-177  
  definiție, 160  
  exemple de ~, 408-411  
  folosire cu tipuri multiple, 165-168  
  localizare, 168

## T

tastatură I/O, 1-33  
tellig (funcție membru), 94  
tellop (funcție membru), 94  
template (cuvânt-cheie), 160  
testarea rezultatelor  
  cu I/O pe fișier, 84-86  
  operații pe stream, 36-41  
throw (cuvânt-cheie), 336  
tie (funcție membru ios), 293-295

## Index

## Z

zecimal, afișări valori în ~, 9-12  
zonă de memorie alocabilă, *vezi și spațiu liber*, 198  
  colectare resturi, 371-373  
  efectele alocării memoriei asupra ~, 359-366  
  gestiunea ~, 359-373  
  listă cu legături, 362-364  
  testarea validității, 367-373  
  intrare specifică, 368-369  
  spațiu liber, 369  
  stare, 367-368

tilda (~), 59

try (cuvânt-cheie), 336

## U

uniuni  
  clase comparate cu ~, 45  
unsetf (funcție membru), 32-33

## V

variabile  
  care partajează un membru al clasei, 248-249  
  durata de viață a ~, 252-254  
  externe, 245-246  
  inițializare, 246-247  
  localizarea declarațiilor, 231  
vertical (caracterul tabulare \v), 7  
virgulă mobilă, precizie de afișare în ~, 15-17  
virtual, destructor ~, 330-333  
virtuale, funcții ~, 209-226, 323-333  
  destructori, 330-333  
  legare dinamică, 327  
  moștenire pe mai multe niveluri, 221-226  
  pure, 224-226  
  tabelă de, 328-330

## W

write (funcție membru), 86-97